A Formal Approach of Safe Destination-Passing Style Programming for Functional Languages

Thomas Bagrel

April, 2025

Abstract

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

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

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

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

Acknowledgements

TBD.

Contents

Chapter 1

Introduction

1.1 Memory management: a core design axis for programming languages

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Functional programming languages

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

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

- supports lambda abstractions, aka. anonymous functions, as first-class values, allowing functions to be stored, passed as parameters, and to capture parts of their parent environment (closures);

- emphasizes expressions over statements, where each instruction produces a value of a specific type;

- builds complex expressions by applying and composing functions rather than chaining statements.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In Minamide's work, incomplete structures, or *structures with a hole*, are represented by hole abstractions — essentially pure functions that take the missing component of the structure as an argument and return the completed structure. In other terms, it represents the pending construction of a structure, than can theoretically only be carried out when all its missing pieces have been supplied. Hole abstractions can also be composed, like functions: $(\lambda x \mapsto 1 :: x) \circ (\lambda y \mapsto 2 :: y) \rightsquigarrow \lambda y \mapsto 1 :: 2 :: y$. 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 inseparable, which limits an incomplete structure to having only a single hole. For these reasons, I would argue that Minamide's approach does not yet qualify as destination-passing style programming. Nonetheless, it is one of the earliest examples of a pure functional language allowing data structures to be passed with write permissions while preserving key language guarantees such as memory safety and purity.

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

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

1.5 Unifying and formalizing Functional DPS frameworks

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

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

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

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

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

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

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

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

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

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

Chapter 2

Linear λ-calculus

2.1 From λ-calculus to linear λ-calculus

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

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

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

In that sense, Girard first introduced Linear Logic, in 1987, and only later studied the corresponding calculus, aka. linear λ-calculus. Linear logic follows from the observation that in sequent calculus[1], hypotheses are duplicated or discarded using explicit rules of the system, named *contraction* and *weakening*, in contrast to natural deduction where all that happens implicitly, as it is part of the meta-theory. As a result, it is possible to track the number of times a formula is used by counting the use of these structural rules. Linear logic take this idea further, and deliberately restrict contraction and weakening, so by default, every hypothesis must be used exactly once. Consequently, logical implication $\mathtt{T} \to \mathtt{U}$ is not part of linear logic, but is replaced by linear implication $\mathtt{T} \multimap \mathtt{U}$, where $\mathtt{T}$ must be used exactly once to prove $\mathtt{U}$. Linear logic also introduces a modality $!$, pronounced *of course* or *bang*, to allow weakening and contraction on specific formulas: $!\mathtt{T}$ denotes that $\mathtt{T}$ can be used an arbitrary number of times (we say it is *unrestricted*). We say that linear logic is a *substructural* logic because it restricts the use of structural rules of usual logic.

We present in Figure 2.1 the natural deduction formulation of intuitionistic linear logic (ILL), as it lends itself well to a computational interpretation as a linear λ-calculus with usual syntax. We borrow the *sequent style* notation of sequent calculus for easier transition into typing rules of terms later. However, as we are in an intuitionistic setting, rules only derive a single conclusion from a multiset of formulas.

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

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

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

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

$$\frac{\Gamma_1 \vdash \mathtt{1} \qquad \Gamma_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \mathrm{ILL}/\mathtt{1}\mathrm{E} \qquad \frac{\Gamma_1 \vdash \mathtt{T}_1 \oplus \mathtt{T}_2 \qquad \Gamma_2,\ \mathtt{T}_1 \vdash \mathtt{U} \qquad \Gamma_2,\ \mathtt{T}_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \mathrm{ILL}/{\oplus}\mathrm{E} \qquad \frac{\Gamma_1 \vdash \mathtt{T}_1 \otimes \mathtt{T}_2 \qquad \Gamma_2,\ \mathtt{T}_1,\ \mathtt{T}_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \mathrm{ILL}/{\otimes}\mathrm{E} \qquad \frac{!\Gamma \vdash \mathtt{T}}{!\Gamma \vdash !\mathtt{T}}\ \mathrm{ILL}/!\mathrm{P}$$

$$\frac{\Gamma \vdash !\mathtt{T}}{\Gamma \vdash \mathtt{T}}\ \mathrm{ILL}/!\mathrm{D} \qquad \frac{\Gamma_1 \vdash !\mathtt{T} \qquad \Gamma_2,\ !\mathtt{T},\ !\mathtt{T} \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \mathrm{ILL}/!\mathrm{C} \qquad \frac{\Gamma_1 \vdash !\mathtt{T} \qquad \Gamma_2 \vdash \mathtt{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathtt{U}}\ \mathrm{ILL}/!\mathrm{W}$$

Figure 2.1: Natural deduction formulation of intuitionistic linear logic (sequent-style)

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

Figure 2.2: Grammar of linear λ-calculus in monadic presentation (λ_{L_1})

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

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

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

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

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

There exists several possible interpretations of linear logic as a linear λ-calculus. The first one, named *monadic* presentation of linear λ-calculus in [AND92], and denoted λ_{L_1} in this document, is a direct term assignment of the natural deduction rules of ILL given in Figure 2.1. The syntax and typing rules of this presentation, inspired greatly from the work of [Bie94], are given in Figures 2.2 and 2.3.

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

$$\frac{}{x : \mathsf{T} \vdash x : \mathsf{T}}\ \lambda_{L_1}/\mathrm{Id} \qquad \frac{\Gamma,\ x : \mathsf{T} \vdash u : \mathsf{U}}{\Gamma \vdash \lambda x \mapsto u : \mathsf{T} \multimap \mathsf{U}}\ \lambda_{L_1}/{\multimap}\mathrm{I} \qquad \frac{}{\bullet \vdash () : \mathsf{1}}\ \lambda_{L_1}/\mathsf{1}\mathrm{I}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{L_1}/{\oplus}\mathrm{I}_1 \qquad \frac{\Gamma \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{L_1}/{\oplus}\mathrm{I}_2 \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2 \vdash t_2 : \mathsf{T}_2}{\Gamma_1,\ \Gamma_2 \vdash (t_1\,,\, t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \lambda_{L_1}/{\otimes}\mathrm{I}$$

$$\frac{\Gamma_1 \vdash t : \mathsf{T} \quad \Gamma_2 \vdash t' : \mathsf{T} \multimap \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash t'\ t : \mathsf{U}}\ \lambda_{L_1}/{\multimap}\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : \mathsf{1} \quad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ \lambda_{L_1}/\mathsf{1}\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 : \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \lambda_{L_1}/{\oplus}\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1,\ x_2 : \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1\,,\, x_2) \mapsto u : \mathsf{U}}\ \lambda_{L_1}/{\otimes}\mathrm{E} \qquad \frac{!\Gamma \vdash t : \mathsf{T}}{!\Gamma \vdash \mathsf{Many}\, t : !\mathsf{T}}\ \lambda_{L_1}/!\mathrm{P} \qquad \frac{\Gamma \vdash t : !\mathsf{T}}{\Gamma \vdash \mathbf{derelict}\, t : \mathsf{T}}\ \lambda_{L_1}/!\mathrm{D}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : !\mathsf{T} \\ \Gamma_2,\ x_1 : !\mathsf{T},\ x_2 : !\mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathbf{dup}\ t\ \mathsf{as}\ x_1\,,\ x_2\ \mathsf{in}\ u : \mathsf{U}}\ \lambda_{L_1}/!\mathrm{C} \qquad \frac{\Gamma_1 \vdash t : !\mathsf{T} \quad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathbf{drop}\ t\ \mathsf{in}\ u : \mathsf{U}}\ \lambda_{L_1}/!\mathrm{W}$$

Figure 2.3: Typing rules for linear λ-calculus in monadic presentation (λ_{L_1})

In λ_{L_1}, Γ is now a finite map of variables to types, that can be represented as a set of variable bindings $x : \mathsf{T}$. As usual, duplicated variable names are not allowed in a context Γ. The comma operator denotes disjoint union for finite maps.

Term grammar for λ_{L_1} borrows most of simply typed lambda calculus grammar. In addition, the language has a data constructor/wrapper for unrestricted terms and values, denoted by $\mathsf{Many}\, t$ and $\mathsf{Many}\, v$. Elimination of unit type $\mathsf{1}$ is made with $\fatsemi$ operator. Pattern-matching on sum and product types is made with the **case** keyword. Finally, we have new operators **dup**, **drop** and **derelict** for respective contraction, weakening and dereliction of unrestricted terms of type $!\mathsf{T}$. Promotion of a term to an unrestricted form is made by direct application of constructor Mod . For easier presentation, we also introduce syntactic sugar $\mathbf{let}\ x := t\ \mathbf{in}\ u$ and encode it as $(\lambda x \mapsto u)\ (t)$.

2.3 Implicit structural rules for unrestricted resources

In the monadic presentation λ_{L_1}, the use of unrestricted terms can become very verbose and unhandy because of the need for explicit contraction, weakening and dereliction. A second and equivalent presentation of our linear λ-calculus, named *dyadic* presentation or λ_{L_2}, tend to alleviate this issue by using two typing contexts on each judgment, one for linear variables and one for unrestricted variables. The syntax and typing rules of λ_{L_2} are given in Figures 2.4 and 2.5.

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

Figure 2.4: Grammar of linear λ-calculus in dyadic presentation (λ_{L_2})

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

$$\frac{}{x : \mathsf{T} \,;\, \mho \vdash x : \mathsf{T}}\ \lambda_{L_2}/\mathrm{Id}_{\mathrm{Lin}} \qquad \frac{}{\bullet \,;\, \mho,\, x : \mathsf{T} \vdash x : \mathsf{T}}\ \lambda_{L_2}/\mathrm{Id}_{\mathrm{Ur}} \qquad \frac{\Gamma,\, x : \mathsf{T} \,;\, \mho \vdash u : \mathsf{U}}{\Gamma \,;\, \mho \vdash \lambda x \mapsto u : \mathsf{T} \multimap \mathsf{U}}\ \lambda_{L_2}/{\multimap}\mathrm{I}$$

$$\frac{}{\bullet \,;\, \mho \vdash () : \mathsf{1}}\ \lambda_{L_2}/\mathsf{1}\mathrm{I} \qquad \frac{\Gamma \,;\, \mho \vdash t_1 : \mathsf{T}_1}{\Gamma \,;\, \mho \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{L_2}/{\oplus}\mathrm{I}_1 \qquad \frac{\Gamma \,;\, \mho \vdash t_2 : \mathsf{T}_2}{\Gamma \,;\, \mho \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \lambda_{L_2}/{\oplus}\mathrm{I}_2$$

$$\frac{\Gamma_1 \,;\, \mho \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2 \,;\, \mho \vdash t_2 : \mathsf{T}_2}{\Gamma_1,\, \Gamma_2 \,;\, \mho \vdash (t_1 , t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \lambda_{L_2}/{\otimes}\mathrm{I} \qquad \frac{\bullet \,;\, \mho \vdash t : \mathsf{T}}{\bullet \,;\, \mho \vdash \mathsf{Many}\, t : \,!\mathsf{T}}\ \lambda_{L_2}/\,!\mathrm{I} \qquad \frac{\Gamma_1 \,;\, \mho \vdash t : \mathsf{T} \quad \Gamma_2 \,;\, \mho \vdash t' : \mathsf{T} \multimap \mathsf{U}}{\Gamma_1,\, \Gamma_2 \,;\, \mho \vdash t'\, t : \mathsf{U}}\ \lambda_{L_2}/{\multimap}\mathrm{E}$$

$$\frac{\Gamma_1 \,;\, \mho \vdash t : \mathsf{1} \quad \Gamma_2 \,;\, \mho \vdash u : \mathsf{U}}{\Gamma_1,\, \Gamma_2 \,;\, \mho \vdash t \fatsemi u : \mathsf{U}}\ \lambda_{L_2}/\mathsf{1}\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \,;\, \mho \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\, x_1 : \mathsf{T}_1 \,;\, \mho \vdash u_1 : \mathsf{U} \\ \Gamma_2,\, x_2 : \mathsf{T}_2 \,;\, \mho \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1,\, \Gamma_2 \,;\, \mho \vdash \mathbf{case}\ t\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1 , \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \lambda_{L_2}/{\oplus}\mathrm{E}$$

$$\frac{\Gamma_1 \,;\, \mho \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \quad \Gamma_2,\, x_1 : \mathsf{T}_1,\, x_2 : \mathsf{T}_2 \,;\, \mho \vdash u : \mathsf{U}}{\Gamma_1,\, \Gamma_2 \,;\, \mho \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1 , x_2) \mapsto u : \mathsf{U}}\ \lambda_{L_2}/{\otimes}\mathrm{E} \qquad \frac{\Gamma_1 \,;\, \mho \vdash t : \,!\mathsf{T} \quad \Gamma_2 \,;\, \mho,\, x : \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_1,\, \Gamma_2 \,;\, \mho \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u : \mathsf{U}}\ \lambda_{L_2}/\,!\mathrm{E}$$

Figure 2.5: Typing rules for linear λ-calculus in dyadic presentation (λ_{L_2})

In λ_{L_2} there is no longer rules for contraction, weakening, and dereliction of unrestricted resources. Instead, each judgment is equipped with a second context $\mho$ that holds variable bindings that can be used in an unrestricted fashion. The $\mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u$ construct is used to bind a term of type $!\mathsf{T}$ as a variable binding $x : \mathsf{T}$ in the unrestricted context $\mho$. It's important to note that $\mathbf{case}\ t\ \mathbf{of}\ \mathsf{Many}\, x \mapsto u$ is not dereliction: one can still use x several times within body u, or recreate t by wrapping x back as $\mathsf{Many}\, x$; while that wouldn't be possible in $\mathbf{let}\, x := \mathbf{derelict}\, t\ \mathbf{in}\, u$ of λ_{L_1}. Morally, we can view the pair of contexts $\Gamma ; \mho$ of λ_{L_2} as a single context $\Gamma,\, !\mho$ of λ_{L_1}, where $!\mho$ is the context with the same variable bindings as $\mho$, except that all types are prefixed by $!$.

In λ_{L_2}, contraction for unrestricted resources happens implicitly every time a rule has two subterms as premises. Indeed, the unrestricted context $\mho$ is duplicated in both premises, unlike the linear context Γ that must be split into two disjoint parts. All unrestricted variable bindings are thus propagated to the leaves of the typing tree, that is, the rules with no premises $\lambda_{L_2}/\mathrm{Id}_{\mathrm{Lin}}$, $\lambda_{L_2}/\mathrm{Id}_{\mathrm{Ur}}$, and $\lambda_{L_2}/\mathsf{1}\mathrm{I}$. These three rules discard all bindings of the unrestricted context $\mho$ that aren't used, performing several implicit weakening steps. Finally, this system has two identity rules. The first one, $\lambda_{L_2}/\mathrm{Id}_{\mathrm{Lin}}$, is the usual linear identity: it asks for the variable x to be in the linear typing context, and later x cannot be reused in another subterm. The second one, $\lambda_{L_2}/\mathrm{Id}_{\mathrm{Ur}}$, is the unrestricted identity: it lets us use the variable x from the unrestricted typing context in a place where a linear variable is expected, performing a sort of implicit dereliction.

[AND92] has a detailed proof that both λ_{L_1} and λ_{L_2} are equivalent, in other terms, that a program t types in the pair of contexts $\Gamma; \mathcal{U}$ in λ_{L_2} if and only if it types in context $\Gamma,\ ?\mathcal{U}$ in λ_{L_1}.

2.4 Back to a single context with the graded modal approach

In [GSS92], Girard introduced a *bounded* version of linear logic, where the $!$ modality is accompanied by an index m that specifies how many times an hypothesis can be used. This *graded* modality $!_{\mathsf{m}}$ gives more flexibility than the binary choice we had before, where resources could either be strictly linear or fully unrestricted. The index m — that we will call *mode* from now on — takes its values in an arbitrary set equipped with a semiring or ringoid structure (it can be integers for precise use count; extended integers to take unrestricted use into account; or even intervals to represent minimum and maximum uses).

Having an arbitrary number of graded modalities $(!_{\mathsf{m}})_{\mathsf{m}\in\mathsf{M}}$ 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.

Since [GSS92], the line of work of graded modalities has been fruitfully developed, but most bounded/graded linear calculi kept a presentation close to the monadic presentation λ_{L_1}, in which there is a clear distinction between linear resources and modality-annotated ones, and modality-annotated resources are kept "boxed" with their modality $!_{\mathsf{m}}$ inside the typing context Γ, and are only "deboxed" at the latest, through an explicit use of the dereliction rule.

However, that changed in 2014, when both [GS14] and [POM14] introduced calculi where judgments have a single typing context, in which *all* variable bindings carry a mode m representing how they must be used. For instance, linear bindings $x : \mathsf{T}$ are replaced by modal binding requiring one use $x :_{1} \mathsf{T}$. As a result, thanks to the variable bindings enriched with modes, modality-annotated resources $t : !_{\mathsf{m}}\mathsf{T}$ can be deboxed very early — as in the dyadic presentation — in the form of a binding $x :_{\mathsf{m}} \mathsf{T}$, without loosing any information and flexibility. We recover a system in which contraction, weakening, and dereliction can be made conveniently implicit, without loosing any control power over resource use! For that, we just have to define a few operators on *modes*, that we will then lift to variable bindings, and then to typing contexts themselves.

In the spirit of [GS14] and subsequent [Ber+18; AB20], we'll now show what the concrete *modal* presentation for intuitionistic λ-calculus or λ_{L_m} looks like. We will take a very simple ringoid for modes, that is enough to model linearity equivalently as the previous presentations λ_{L_1} and λ_{L_2}: 1 for variables that must be managed in strict linear fashion, and ω for unrestricted ones. We give the following operation tables:

$+$	1	ω
1	ω	ω
ω	ω	ω

$\cdot$	1	ω
1	1	∞
ω	∞	∞

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

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

Equipped with those, we can move to the grammar and typing rules of λ_{L_m}, given in Figures 2.6 and 2.7. Constructor for $!_{\mathsf{m}}\mathsf{T}$ is now $\mathsf{Mod}_{\mathsf{n}}\, t$ (instead of $\mathsf{Many}\, t$ for $!\mathsf{T}$).

In λ_{L_m} we're back to a single identity rule λ_{L_m} / Id, that asks for x to be in the typing context with a mode m that must be *compatible with a single linear use* (we note $1 \lesssim \mathsf{m}$). In our ringoid, with only two elements, we have both $1 \lesssim 1$ and $1 \lesssim \omega$, so x can actually have any mode (either linear or unrestricted, so encompassing both λ_{L_2} / $\mathrm{Id}_{\mathrm{Lin}}$ and λ_{L_2} / $\mathrm{Id}_{\mathrm{Ur}}$), but in more complex modal systems, not all modes have to be compatible with the unit of the ringoid[2]. Rules λ_{L_m} / Id and λ_{L_m} / 1I also allows to discard any context composed only of unrestricted bindings, denoted by $\omega\cdot\Gamma$ (equivalent to notation $!\Gamma$ in λ_{L_1}), so they are performing implicit weakening as in λ_{L_2}.

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

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

Figure 2.6: Grammar of linear λ-calculus in modal presentation (λ_{L_m})

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

$$\frac{\mathtt{1} \leqslant \mathtt{m}}{\omega\cdot\Gamma,\; x :_{\mathtt{m}} \mathtt{T} \vdash x : \mathtt{T}}\;\lambda_{L_m}/\mathrm{Id} \qquad \frac{\Gamma,\; x :_{\mathtt{m}} \mathtt{T} \vdash u : \mathtt{U}}{\Gamma \vdash \lambda x\,{}_{\mathtt{m}}\!\mapsto u : \mathtt{T}\,{}_{\mathtt{m}}\!\multimap \mathtt{U}}\;\lambda_{L_m}/{\multimap}\mathrm{I} \qquad \frac{}{\omega\cdot\Gamma \vdash () : \mathtt{1}}\;\lambda_{L_m}/\mathtt{1}\mathrm{I}$$

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

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

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathtt{T}_1 \oplus \mathtt{T}_2 \\ \Gamma_2,\; x_1 :_{\mathtt{1}} \mathtt{T}_1 \vdash u_1 : \mathtt{U} \\ \Gamma_2,\; x_2 :_{\mathtt{1}} \mathtt{T}_2 \vdash u_2 : \mathtt{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{case}\;t\;\mathtt{of}\;\{\mathtt{Inl}\,x_1 \mapsto u_1\,,\;\mathtt{Inr}\,x_2 \mapsto u_2\} : \mathtt{U}}\;\lambda_{L_m}/{\oplus}\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathtt{T}_1 \otimes \mathtt{T}_2 \\ \Gamma_2,\; x_1 :_{\mathtt{1}} \mathtt{T}_1,\; x_2 :_{\mathtt{1}} \mathtt{T}_2 \vdash u : \mathtt{U}\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{case}\;t\;\mathtt{of}\;(x_1\,,\,x_2) \mapsto u : \mathtt{U}}\;\lambda_{L_m}/{\otimes}\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : !_{\mathtt{n}}\mathtt{T} \quad \Gamma_2,\; x :_{\mathtt{n}} \mathtt{T} \vdash u : \mathtt{U}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{case}\;t\;\mathtt{of}\;\mathtt{Mod}_{\mathtt{n}}\,x \mapsto u : \mathtt{U}}\;\lambda_{L_m}/!\mathrm{E}$$

Figure 2.7: Typing rules for linear λ-calculus in modal presentation (λ_{L_m})

Every rule of λ_{L_m} 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 $\mathtt{1}$ or ω, it doesn't matter), then $\Gamma_1 + \Gamma_2$ will contain binding $x :_{\omega} \mathtt{T}$. Said differently, the parent term will automatically deduce wether x needs to be linear or unrestricted based on how (many) subterms use x, thanks to the $+$ operator. It's a vastly different approach than the linear context split for subterms in $\lambda_{L_{\{1,2\}}}$ and unrestricted context duplication for subterms in λ_{L_2}. I would argue that it makes the system smoother, and allows for easy extension without changing the typing rules much (compared to the monadic / dyadic presentation that are very specialized to handle linearity/unrestrictedness, and that alone).

Much as in λ_{L_2}, the exponential modality $!_{\mathtt{n}}$ is eliminated by a $\mathtt{case}\;t\;\mathtt{of}\;\mathtt{Mod}_{\mathtt{n}}\,x \mapsto u$ expression, that binds the payload to a new variable x with modality $\mathtt{n}$. The original boxed value can be recreated if needed with $\mathtt{Mod}_{\mathtt{n}}\,x$; indeed, as in λ_{L_2}, elimination for $!_{\mathtt{n}}$ is *not* dereliction.

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

2.5 Deep modes: projecting modes through fields of data structures

In original linear logic from Girard (see ILL and presentation λ_{L_1} above), there is only a one-way morphism between $!\mathtt{T}\otimes!\mathtt{U}$ and $!(\mathtt{T}\otimes\mathtt{U})$; we cannot go from $!(\mathtt{T}\otimes\mathtt{U})$ to $!\mathtt{T}\otimes!\mathtt{U}$. In other terms, an unrestricted pair $!(\mathtt{T}\otimes\mathtt{U})$ doesn't allow for unrestricted use of its components. The pair can be duplicated or discarded at will, but to be used, it needs to be derelicted first, to become $\mathtt{T}\otimes\mathtt{U}$, that no longer allow to duplicate or discard any of $\mathtt{T}$ or $\mathtt{U}$. As a result, $\mathtt{T}$ and $\mathtt{U}$ will have to be used exactly the same number of times, even though they are part of an unrestricted pair!

Situation is no different in λ_{L_2} (resp. λ_{L_m}): although the pair of type $!(\mathtt{T}\otimes\mathtt{U})$ can be bound in an unrestricted binding $x : \mathtt{T}\otimes\mathtt{U} \in \mathcal{U}$ (resp. $x :_\omega \mathtt{T}\otimes\mathtt{U}$) and be used as this without need for dereliction, when it will be pattern-matched on with $\lambda_{L_{\{2,m\}}}$ / $\otimes$E, implicit dereliction will still happen, and linear-only bindings will be made for its components: $x_1 : \mathtt{T},\ x_2 : \mathtt{U}$ in the linear typing context (resp. $x_1 :_1 \mathtt{T},\ x_2 :_1 \mathtt{U}$).

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

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

$$\begin{aligned}&\mathtt{fst_{dm}} : \mathtt{T}\otimes\mathtt{U} \multimap_\omega \mathtt{T}\\ &\mathtt{fst_{dm}} \triangleq \lambda x \mapsto_\omega \mathtt{case}_\omega\ x\ \mathtt{of}\ (x_1, x_2) \mapsto x_1\end{aligned}$$

as we would do in a non-linear language, like STLC, in which `fst` would have type $\mathtt{T}\otimes\mathtt{U} \to \mathtt{T}$. On the other hand, in λ_{L_m} as presented in Figures 2.6 and 2.7, we need to indicate individually for each component of the pair that we want them unrestricted (at least for the second one that we intent on discarding), and add extra elimination for the $!$ modality:

$$\begin{aligned}&\mathtt{fst_m} : \mathtt{T}\otimes(!_\omega\mathtt{U}) \multimap_1 \mathtt{T}\\ &\mathtt{fst_m} \triangleq \lambda x \mapsto_1 \mathtt{case}\ x\ \mathtt{of}\ (x_1, ux_2) \mapsto\\ &\qquad\qquad \mathtt{case}\ ux_2\ \mathtt{of}\ \mathtt{Mod}_\omega\ x_2 \mapsto x_1\end{aligned}$$

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

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

The only change needed on the grammar is that `case` constructs now take a mode $\mathtt{m}$ to which they consume the scrutinee, that is propagated to the field(s) of the scrutinee in the body of the `case`. The new typing rules for `case` are presented in Figure 2.8, all the other rules of linear λ-calculus with deep modes, $\lambda_{L_{dm}}$, are supposed identical to Figure 2.7.

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

$$\frac{\Gamma_1 \vdash t : \mathtt{T}_1 \oplus \mathtt{T}_2 \quad \Gamma_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1 \vdash u_1 : \mathtt{U} \quad \Gamma_2,\ x_2 :_{\mathtt{m}} \mathtt{T}_2 \vdash u_2 : \mathtt{U}}{\mathtt{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathtt{case}_{\mathtt{m}}\ t\ \mathtt{of}\ \{\mathtt{Inl}\, x_1 \mapsto u_1\ ,\ \mathtt{Inr}\, x_2 \mapsto u_2\} : \mathtt{U}}\ \lambda_{L_{dm}} / \oplus\mathrm{E}$$

$$\frac{\Gamma_1 \vdash t : \mathtt{T}_1 \otimes \mathtt{T}_2 \quad \Gamma_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1,\ x_2 :_{\mathtt{m}} \mathtt{T}_2 \vdash u : \mathtt{U}}{\mathtt{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathtt{case}_{\mathtt{m}}\ t\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto u : \mathtt{U}}\ \lambda_{L_{dm}} / \otimes\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : !_{\mathtt{n}}\mathtt{T} \quad \Gamma_2,\ x :_{\mathtt{m}\cdot\mathtt{n}} \mathtt{T} \vdash u : \mathtt{U}}{\mathtt{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathtt{case}_{\mathtt{m}}\ t\ \mathtt{of}\ \mathtt{Mod}_{\mathtt{n}}\ x \mapsto u : \mathtt{U}}\ \lambda_{L_{dm}} / !\mathrm{E}$$

Figure 2.8: Altered typing rules for deep modes in linear λ-calculus in modal presentation ($\lambda_{L_{dm}}$)

We observe that the typing context Γ_1 in which the scrutinee types is scaled by $\mathtt{m}$ in the conclusion of all the **case** rules. This is very analog to the application rule $\lambda_{L_m} / \multimap\mathrm{E}$: it makes sure that the resources required to type t are consumed $\mathtt{m}$ times if we want to use t at mode $\mathtt{m}$. Given that $x_2 :_1 \mathtt{T}_2 \vdash ((), x_2) : \mathtt{1} \otimes \mathtt{T}_2$ (the pair uses x_2 exactly once), if we want to extract the pair components with mode ω to drop x_2 (as in $\mathtt{fst}_{\mathtt{dm}}$), then $\mathtt{case}_{\omega}\ ((), x_2)\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto x_2$ will require context $\omega \cdot (x_2 :_1 \mathtt{T}_2)$ i.e. $x_2 :_{\omega} \mathtt{T}_2$. In other terms, we cannot use parts of a structure made using linear resources in an unrestricted way (as that would break linearity).

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

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

Many semantics presentations exist for lambda calculus. In Figure 2.9 I present a small-step reduction system for $\lambda_{L_{dm}}$, in *reduction semantics* style inspired from [Fel87] and subsequent [DN04; BD07], that is, a semantics in which the evaluation context E is manipulated explicitly and syntactically as a stack.

Small-step evaluation rules are of three kinds:

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

To have a fully deterministic and and straightforward reduction system, focusing rules can only trigger when the subterm in question is not already a value (denoted by $\star$ in Figure 2.9).

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

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

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

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

$\star$: only allowed if t is not already a value

Figure 2.9: Small-step semantics for $\lambda_{L_{dm}}$

Chapter 3

Formal functional language with first-class destinations: λ_d

3.1 Introduction

In destination-passing style, a function doesn't return a value: it takes as an argument a location where the value ought to be returned. In our notation, a function of type $\mathtt{T} \multimap \mathtt{U}$ would, in destination-passing style, have type $\mathtt{T} \multimap \lfloor \mathtt{U} \rfloor \multimap \mathtt{1}$ instead. This style is common in systems programming, where destinations $\lfloor \mathtt{U} \rfloor$ are more commonly known as "out parameters". In C, $\lfloor \mathtt{U} \rfloor$ would typically be a pointer of type $\mathtt{U}*$.

The reason why system programs rely on destinations so much is that using destinations can save calls to the memory allocator. If a function returns a $\mathtt{U}$, it has to allocate the space for a $\mathtt{U}$. But with destinations, the caller is responsible for finding space for a $\mathtt{U}$. The caller may simply ask for the space to the memory allocator, in which case we've saved nothing; but it can also reuse the space of an existing $\mathtt{U}$ which it doesn't need anymore, or it could use a space in an array, or it could allocate the space in a region of memory that the memory allocator doesn't have access to, like a memory-mapped file.

This does all sound quite imperative, but we argue that the same considerations are relevant for functional programming, albeit to a lesser extent. In fact [Sha+17] has demonstrated that using destination passing in the intermediate language of a functional array-programming language allowed for some significant optimizations. Where destinations truly shine in functional programming, however, is that they increase the expressiveness of the language; destinations as first-class values allow for meaningfully new programs to be written. This point was first explored in [Bag24].

The trouble, of course, is that destinations are imperative; we wouldn't want to sacrifice the immutability of our linked data structure (we'll usually just say "structure") for the sake of the more situational destinations. The goal is to extend functional programming just enough to be able to build immutable structures by destination passing without endangering purity and memory safety. This is precisely what [Bag24] does, using a linear type system to restrict mutation. Destinations become write-once references into an immutable structure with holes. In that we follow their leads, but we refine the type system further to allow for even more programs, as we discuss in Section 3.3.

There are two key elements to the expressiveness of destination passing:

- 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 program using function composition in a form of continuation-passing; and destinations act as an optimization. This line of work was pioneered by [Min98]. While this only increases expressiveness when combined with the next point, the optimization is significant enough that destination passing has been implemented in the Ocaml optimizer to support tail modulo constructor [BCS21];

- when destinations are first-class values, they can be passed and stored like ordinary values. This is the innovation of [Bag24] upon which we build. The consequence is that not only the order in which a structure is built is arbitrary, this order can be determined dynamically during the runtime of the program.

To support this programming style, we introduce λ_d. We intend λ_d to serve as a core calculus to reason about safe destinations. Indeed λ_d subsumes all the systems that we've discussed in this section: they can all be encoded in λ_d via simple macro expansion. As such we expect that potential extensions to these systems can be justified by giving their semantics as an expansion in λ_d.

Our contributions are as follows:

- λ_d, a linear and modal simply typed λ-calculus with destinations (Sections 3.5 and 3.6). λ_d is expressive enough so that previous calculi for destinations can be encoded in λ_d (see Section 3.9);
- a demonstration that λ_d is more expressive than previous calculi with destinations (Sections 3.3 and 3.4), namely that destinations can be stored in structures with holes. We show how we can improve, in particular, on the breadth-first traversal example of [Bag24];
- an implementation strategy for λ_d which uses mutation without compromising the purity of λ_d (Section 4.5);
- formally-verified proofs, with the Coq proof assistant, of the main safety lemmas (Section 3.7).

3.2 Working with destinations

Let's introduce and get familiar with λ_d, our simply typed λ-calculus with destination. The syntax is standard, except that we use linear logic's $\mathtt{T} \oplus \mathtt{U}$ and $\mathtt{T} \otimes \mathtt{U}$ for sums and products, since λ_d is linearly typed, even though it isn't a focus in this section.

Building up a vocabulary

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

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

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

The form $d \blacktriangleleft x$ is the simplest way to use a destination. But we don't have to fill a destination with a complete value in a single step. Destinations can be filled piecemeal.

$$\begin{aligned} &\mathtt{fillWithInlCtor} : \lfloor \mathtt{T} \oplus \mathtt{U} \rfloor \multimap \lfloor \mathtt{T} \rfloor \\ &\mathtt{fillWithInlCtor}\ d \triangleq d \triangleleft \mathtt{Inl} \end{aligned}$$

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

Notice that with **fillWithInlCtor** we are constructing the structure from the outermost constructor inward: we've written a value of the form $\mathtt{Inl}\ \square$ 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*. This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we make a value v and only then can we use $\mathtt{Inl}$ to make an $\mathtt{Inl}\ v$. This will turn out to be a key ingredient in the expressiveness of destination passing.

Yet, everything we've shown so far could have been done with continuations. So it's worth asking: how are destination different from continuations? Part of the answer lies in our intention to effectively implement destinations as pointers to uninitialized memory (see Section 4.5). But where destinations really differ from continuations is when one has several destinations at hand. Then they have to fill *all* the destinations; whereas when one has multiple continuations, they can only return to one of them. Multiple destination arises when a destination of pair gets filled with a hollow pair constructor:

$$\begin{aligned}&\texttt{fillWithPairCtor} : \lfloor \mathtt{T} \otimes \mathtt{U} \rfloor \multimap \lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{U} \rfloor\\ &\texttt{fillWithPairCtor}\ d \triangleq d \triangleleft (,)\end{aligned}$$

After using `fillWithPairCtor`, the user must fill both the first field *and* the second field, using the destinations of type $\lfloor \mathtt{T} \rfloor$ and $\lfloor \mathtt{U} \rfloor$ respectively. In plain English, it sounds obvious, but the key remark is that `fillWithPairCtor` doesn't exist on continuations.

Structures with holes It is crucial to note that while a destination is used to build a structure, the type of the structure being built might be different from the type of the destination that is being filled. A destination of type $\lfloor \mathtt{T} \rfloor$ is a pointer to a yet-undefined part of a bigger structure. We say that such a structure has a hole of type $\mathtt{T}$; but the type of the structure itself isn't specified (and never appears in the signature of destination-filling functions). For instance, using `fillWithPairCtor` only indicates that the structure being operated on has a hole of type $\mathtt{T} \otimes \mathtt{U}$ that is being written to.

Thus, we still need a type to tie the structure under construction — left implicit by destination-filling primitives — with the destinations representing its holes. To represent this, λ_d introduces a type $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$ for a structure of type $\mathtt{S}$ missing a value of type $\mathtt{T}$ to be complete. There can be several holes in $\mathtt{S}$, resulting in several destinations on the right hand side: for example, $\mathtt{S} \ltimes (\lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{U} \rfloor)$ represents a $\mathtt{S}$ that misses both a $\mathtt{T}$ and a $\mathtt{U}$ to be complete.

The general form $\mathtt{S} \ltimes \mathtt{T}$ is read "$\mathtt{S}$ ampar $\mathtt{T}$". The name "ampar" stands for "asymmetric memory par"; we will explain why it is asymmetric in Section 3.5. For now, it's sufficient to observe that $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$ is akin to a "par" type $\mathtt{S} \parr \mathtt{T}^{\perp}$ in linear logic; you can think of $\mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor$ as a (linear) function from $\mathtt{T}$ to $\mathtt{S}$. That structures with holes could be seen a linear functions was first observed in [Min98], we elaborate on the value of having a par type rather than a function type in Section 3.4. A similar connective is called `Incomplete` in [Bag24].

Destinations always exist within the context of a structure with holes. A destination is both a witness of a hole present in the structure, and a handle to write to it. Crucially, destinations are otherwise ordinary values. To access the destinations of an ampar, λ_d provides a `map` construction, which lets us apply a function to the right-hand side of the ampar. It is in the body of the `map` construction that functions operating on destinations can be called:

$$\begin{aligned}&\texttt{fillWithInlCtor}' : \mathtt{S} \ltimes \lfloor \mathtt{T} \oplus \mathtt{U} \rfloor \multimap \mathtt{S} \ltimes \lfloor \mathtt{T} \rfloor\\ &\texttt{fillWithInlCtor}'\ x \triangleq \texttt{map}\ x\ \texttt{with}\ d \mapsto \texttt{fillWithInlCtor}\ d\\ &\texttt{fillWithPairCtor}' : \mathtt{S} \ltimes \lfloor \mathtt{T} \otimes \mathtt{U} \rfloor \multimap \mathtt{S} \ltimes (\lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{U} \rfloor)\\ &\texttt{fillWithPairCtor}'\ x \triangleq \texttt{map}\ x\ \texttt{with}\ d \mapsto \texttt{fillWithPairCtor}\ d\end{aligned}$$

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

Equipped with these, we can, for instance, derive traditional constructors from piecemeal filling. In fact, 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.

[1] As the name suggest, there is a more general elimination $\texttt{from}_{\ltimes}$. It will be discussed in Section 3.5.

$$\begin{aligned}&\mathsf{Inl} \mathbin{:} \mathsf{T} \multimap \mathsf{T} \oplus \mathsf{U}\\&\mathsf{Inl}\, x \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto d \lhd \mathsf{Inl} \blacktriangleleft x)\end{aligned}$$

$$\begin{aligned}&(,) \mathbin{:} \mathsf{T} \multimap \mathsf{U} \multimap \mathsf{T} \otimes \mathsf{U}\\&(x\,,\,y) \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \mathbf{case}\ (d \lhd (,))\ \mathbf{of}\ (d_1\,,\,d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y)\end{aligned}$$

Memory safety and purity At this point, the reader may be forgiven for feeling distressed at all the talk of mutations and uninitialized memory. How is it consistent with our claim to be building a pure and memory-safe language? The answer is that it wouldn't be if we'd allow unrestricted use of destination. Instead λ_d uses a linear type system to ensure that:

- destination are written at least once, preventing examples like:

$$\begin{aligned}&\mathbf{forget} \mathbin{:} \mathsf{T}\\&\mathbf{forget} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto ())\end{aligned}$$

  where reading the result of **forget** would result in reading the location pointed to by a destination that we never used, in other words, reading uninitialized memory;
- destination are written at most once, preventing examples like:

$$\begin{aligned}&\mathbf{ambiguous1} \mathbin{:} \mathsf{Bool}\\&\mathbf{ambiguous1} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto d \blacktriangleleft \mathsf{true} \fatsemi d \blacktriangleleft \mathsf{false})\\&\mathbf{ambiguous2} \mathbin{:} \mathsf{Bool}\\&\mathbf{ambiguous2} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \mathbf{let}\ x := (d \blacktriangleleft \mathsf{false})\ \mathbf{in}\ d \blacktriangleleft \mathsf{true} \fatsemi x)\end{aligned}$$

  where **ambiguous1** returns `false` and **ambiguous2** returns `true` due to evaluation order, even though let-expansion should be valid in a pure language.

Functional queues, with destinations

Now that we have an intuition of how destinations work, let's see how they can be used to build usual data structures. For that section, we suppose that λ_d is equipped with equirecursive types and a fixed-point operator, that isn't part of our formally proven fragment.

Linked lists We define lists as the fixpoint of the functor $X \mapsto 1 \oplus (\mathsf{T} \otimes X)$. For convenience, we also define filling operators $\lhd[]$ and $\lhd(::)$:

$$\begin{aligned}&\mathsf{List}\ \mathsf{T} \overset{rec}{\triangleq} 1 \oplus (\mathsf{T} \otimes (\mathsf{List}\ \mathsf{T}))\\&\lhd[] \mathbin{:} \lfloor \mathsf{List}\ \mathsf{T} \rfloor \multimap 1\\&d \lhd [] \triangleq d \lhd \mathsf{Inl} \lhd ()\end{aligned}$$

$$\begin{aligned}&\lhd(::) \mathbin{:} \lfloor \mathsf{List}\ \mathsf{T} \rfloor \multimap \lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{List}\ \mathsf{T} \rfloor\\&d \lhd (::) \triangleq d \lhd \mathsf{Inr} \lhd (,)\end{aligned}$$

Just like we did in Section 3.2 we can recover traditional constructors from filling operators:

$$\begin{aligned}&(::) \mathbin{:} \mathsf{T} \otimes (\mathsf{List}\ \mathsf{T}) \multimap \mathsf{List}\ \mathsf{T}\\&x :: xs \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \mathbf{case}\ (d \lhd (::))\ \mathbf{of}\ (dx\,,\,dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs)\end{aligned}$$

```
DList T ≜ (List T) ⋉ ⌊List T⌋
append : DList T ⊸ T ⊸ DList T
ys append y ≜
    map ys with dys ↦ case (dys ◁ (::)) of
        (dy , dys′) ↦ dy ◀ y ⨟ dys′
concat : DList T ⊸ DList T ⊸ DList T
ys concat ys′ ≜ map ys with d ↦ d ◁∘ ys′
toList : DList T ⊸ List T
toList ys ≜ from′⋉ (map ys with d ↦ d ◁ [])
```

```
Queue T ≜ (List T)⊗(DList T)
singleton : T ⊸ Queue T
singleton x ≜ (Inr (x :: []) , alloc)
enqueue : Queue T ⊸ T ⊸ Queue T
q enqueue y ≜
    case q of (xs , ys) ↦ (xs , ys append y)
dequeue : Queue T ⊸ 1⊕(T⊗(Queue T))
dequeue q ≜
    case q of {
        ((x :: xs) , ys) ↦ Inr (x , (xs , ys)) ,
        ([] , ys) ↦ case (toList ys) of {
            [] ↦ Inl () ,
            x :: xs ↦ Inr (x , (xs , alloc))
        }
    }
```

Figure 3.1: Difference list and queue implementation in equirecursive λ_d

Difference lists Just like in any language, iterated concatenation of lists $((xs_1 +\!\!+ xs_2) +\!\!+ \ldots) +\!\!+ xs_n$ is quadratic in λ_d. The usual solution to this is difference lists. The name difference lists covers many related implementation, but in pure functional languages, a difference list is usually represented as a function [Hug86]. A singleton difference list is $\lambda ys \ \mapsto \ x :: ys$, and concatenation of difference lists is function composition. Difference lists are turned into a list 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.

However, each concatenation 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. We can do better!

In λ_d we can represent a difference list as a list with a hole. A singleton difference list is $x::\Box$. Concatenation is filling the hole with another difference list, using operator $\triangleleft\!\circ$. The details are on the left of Figure 3.1. This encoding makes no superfluous traversal; in fact, concatenation is an $O(1)$ in-place update.

Efficient queue using previously defined structures A simple implementation of a queue in a purely functional language is as a pair of lists $(front\,,\,back)$ [HM81]. Such queues are called *Hood-Melville queues*. Elements are popped from $front$ and are enqueued in $back$. When we need to pop an element and $front$ is empty, then we set the queue to $(\texttt{reverse}\ back\,,\,[])$, and pop from the new front.

For such a 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 of lists that can be extended from the back but only accessed in a single-threaded fashion: difference lists.

So we can give an improved version of the simple functional queue using destinations. This implementation is presented on the right-hand side of Figure 3.1. Note that contrary to an imperative programming language, we can't implement the queue as a single difference list: 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.

3.3 Scope escape of destinations

In Section 3.2, we've silently been making an 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 more subtle than it may first appear.

To see why, let's consider the type $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$: the type of a destination pointing to a hole where a destination is expected. Think of it as an equivalent of the pointer type $\mathsf{T} **$ in the C language. Destinations are indeed ordinary values, so they can be stored in data structures, and before they get effectively stored, holes stand in their places in the structure. In particular if we have $d : \lfloor \mathsf{T} \rfloor$ and $dd : \lfloor\lfloor \mathsf{T} \rfloor\rfloor$, we can form $dd \blacktriangleleft d$.

This, by itself, is fine: we're building a linear data structure containing a destination; the destination d is used linearly as it will eventually be consumed when that structure is consumed. However, as we explained in Section 3.2, destinations always exist within the scope of a structure with holes, and they witness how incomplete the structure is. As a result, to make a structure readable, it is not enough to know that all its remaining destinations will end up on the left of a fill operator *eventually*; they must do so *now*, before the scope they belong to ends. Otherwise some holes might not have been written to *yet* when the structure is made readable.

The problem is, with a malicious use of $dd \blacktriangleleft d$, we can store away d in a parent scope, so d might not end up on the left of a fill operator before the scope it originates from is complete. And still, that would be accepted by the linear type system.

To visualize the problem, let's consider simple linear store semantics. We'll need the $\mathtt{alloc}'$ operator[2]

$$\begin{aligned}&\mathtt{alloc}' : (\lfloor \mathsf{T} \rfloor \multimap 1) \multimap \mathsf{T}\\ &\mathtt{alloc}'\ f \triangleq \mathtt{from}'_{\times}\,(\mathtt{map}\ \mathtt{alloc}\ \mathtt{with}\ d \mapsto f\ d)\end{aligned}$$

The semantics of $\mathtt{alloc}'$ is: allocate a hole in the store, call the function with the corresponding destination; when the function has returned, dereference the destination to obtain a T. Which we can visualize as:

$$\mathcal{S} \mid \mathtt{alloc}'\ (\lambda d \mapsto t) \longrightarrow \mathcal{S} \sqcup \{h := \square\} \mid (t[d := \rightarrow h] \fatsemi \mathtt{deref}\ \rightarrow h)$$

For instance:

$$\begin{aligned}&\{\,\} \mid \mathtt{alloc}'\ (\lambda d \mapsto d \triangleleft \mathtt{Inl} \triangleleft ())\\ \longrightarrow\ &\{h := \square\} \mid \rightarrow h \triangleleft \mathtt{Inl} \triangleleft () \fatsemi \mathtt{deref}\ \rightarrow h\\ \longrightarrow\ &\{h := \mathtt{Inl}\,()\} \mid \mathtt{deref}\ \rightarrow h\\ \longrightarrow\ &\{\,\} \mid \mathtt{Inl}\,()\end{aligned}$$

Now, we are ready to see a counterexample and how it goes wrong:

$$\mathtt{alloc}'\ (\lambda dd \mapsto \mathtt{case}\ (\mathtt{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d))\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\})$$

Here $dd : \lfloor\lfloor \mathtt{Bool} \rfloor\rfloor$ and $d : \lfloor \mathtt{Bool} \rfloor$. The problem with this example stems from the fact that d is fed to dd, instead of being filled, in the scope of d. Let's look a the problem in action:

$$\begin{aligned}&\{\,\} \mid \mathtt{alloc}'\ (\lambda dd \mapsto \mathtt{case}\ (\mathtt{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d))\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\})\\ \longrightarrow\ &\{hd := \square\} \mid \mathtt{case}\ (\mathtt{alloc}'\ (\lambda d \mapsto \rightarrow hd \blacktriangleleft d))\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ \rightarrow hd\\ \longrightarrow\ &\{hd := \square, h := \square\} \mid \mathtt{case}\ (\rightarrow hd \blacktriangleleft \rightarrow h \fatsemi \mathtt{deref}\ \rightarrow h)\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ \rightarrow hd\\ \longrightarrow\ &\{hd := \rightarrow h, h := \square\} \mid \mathtt{case}\ (\mathtt{deref}\ \rightarrow h)\ \mathtt{of}\ \{\mathtt{true} \mapsto ()\,,\ \mathtt{false} \mapsto ()\} \fatsemi \mathtt{deref}\ \rightarrow hd\end{aligned}$$

Because of d escaping its scope, we end up reading uninitialized memory.

[2]The actual semantics that we'll develop in Section 3.6 is more refined and can give a semantics to the more flexible $\mathtt{from}'_{\times}\ t$ and $\mathtt{alloc}$ operators directly, but for now, this simpler semantic will suffice for $\mathtt{alloc}'$.

This example must be rejected by our type system. As demonstrated by [Bag24], it's possible to reject this example using purely a linear type system: they make it so that, in $d \blacktriangleleft t$, t can't be linear. Since all destinations are linear, t can't be, or contain, a destination. This is a rather blunt restriction! Indeed, it makes the type $\lfloor\lfloor \mathtt{T} \rfloor\rfloor$ practically useless: we can form destinations with type $\lfloor\lfloor \mathtt{T} \rfloor\rfloor$ but they can never be filled. More concretely, this means that destination can never be stored in data structures with holes, such as the difference list or queues of Section 3.2.

But we really want to be able to store destinations in data structures with holes, in fact, we'll use this capability to our advantage in Figure 3.2. So we want t in $d \blacktriangleleft t$ to be allowed to be linear. Without further restrictions, the counterexample would be well-typed. To address this, λ_d uses a system of ages to represent scopes. Ages are described in Section 3.5.

3.4 Breadth-first tree traversal

As a more full-fledged example, which uses the full expressive power of λ_d, we borrow and improve on an example from [Bag24], breadth-first tree relabeling:

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

This example cannot be implemented using [Min98] system where structures with holes are represented as linear functions. Destinations as first-class values are very much required. Indeed, breadth-first traversal implies that the order in which the structure must be populated (left-to-right, top-to-bottom) is not the same as the structural order of a functional binary tree, that is, building the leaves first and going up to the root. This isn't very natural in functional programming, which is why a lot has been written on purely functional breadth-first traversals [Oka00; Gib93; Gib+23].

On the other hand, as demonstrated in [Bag24], first-class destination passing lets us use the familiar algorithm with queues that is taught in classroom. Specifically, the algorithm keeps a queue of pairs (*input subtree*, *destination to output*

Figure 3.2 presents the λ_d implementation of the breadth-first tree traversal. There, `Tree T` is defined unsurprisingly as $\mathtt{Tree\ T} \triangleq \mathtt{1} \oplus (\mathtt{T} \otimes ((\mathtt{Tree\ T}) \otimes (\mathtt{Tree\ T})))$; we refer to the constructors of `Tree T` as `Nil` and `Node`, defined in the obvious way. We also assume some encoding of the type `Nat` of natural number. `Queue T` is the efficient queue type from Section 3.2.

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. A similar traversal function can be found in [Bag24]. The difference is that, because they can't store destinations in structures with holes (see the discussion in Section 3.3), their implementation can't use the efficient queue implementation from Section 3.2. So they have to revert to using a Hood-Melville queue for breadth-first traversal.

In **`mapAccumBFS`**, we create a new destination *`dtree`* into which we will write the result of the traversal, then call the main loop **`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 actually use the fact that destinations are ordinary values.

New in Figure 3.2 are the fuchsia annotations: these are *modes*. We'll describe modes in detail in Section 3.5. In the meantime, 1 and ω control linearity: we use ω to mean that the state and function f can be used many times. On the other hand, ∞ is an *age* annotation; in particular, the associated argument cannot carry destinations. Without these modes, our type system presented in Section 3.5 would reject the example. Arguments with no modes are otherwise linear and can capture destinations. We introduce the exponential modality $!_m \mathtt{T}$ to reify mode m in a type; this is useful to return several values having different modes from a function, like in f. An exponential is rarely needed in an argument position, as we have $(!_m \mathtt{T}) \multimap \mathtt{U} \simeq \mathtt{T} \mathbin{_m\!\!\multimap} \mathtt{U}$.

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

Figure 3.2: Breadth-first tree traversal in destination-passing style

$$
\begin{array}{rcl}
t, u & ::= & x \mid t'\ t \mid t \fatsemi t' \\
& \mid & \mathsf{case}_m\ t\ \mathsf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\} \mid \mathsf{case}_m\ t\ \mathsf{of}\ (x_1\ ,\ x_2) \mapsto u \mid \mathsf{case}_m\ t\ \mathsf{of}\ \mathsf{Mod}_n\ x \mapsto u \\
& \mid & \mathsf{map}\ t\ \mathsf{with}\ x \mapsto t' \mid \mathsf{to}_{\ltimes}\ t \mid \mathsf{from}_{\ltimes}\ t \mid \mathsf{alloc} \\
& \mid & t \triangleleft () \mid t \triangleleft \mathsf{Inl} \mid t \triangleleft \mathsf{Inr} \mid t \triangleleft (,) \mid t \triangleleft \mathsf{Mod}_m \mid t \triangleleft (\lambda x\ {}_m\!\mapsto u) \mid t \triangleleft_{\circ} t' \mid t \blacktriangleleft t' \\
\\
\mathsf{T}, \mathsf{U}, \mathsf{S} & ::= & \lfloor_n \mathsf{T} \rfloor \quad \textit{(destination)} \\
& \mid & \mathsf{S} \ltimes \mathsf{T} \quad \textit{(ampar)} \\
& \mid & 1 \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid !_m \mathsf{T} \mid \mathsf{T}\ {}_m\!\multimap \mathsf{U} \\
\\
\mathsf{m}, \mathsf{n} & ::= & \mathsf{pa} \quad \textit{(pair of multiplicity and age)} \\
\mathsf{p} & ::= & 1 \mid \omega \\
\mathsf{a} & ::= & \uparrow^n \mid \infty \\
\\
\Gamma & ::= & \cdot \mid x :_m \mathsf{T} \mid \Gamma_1,\ \Gamma_2 \mid \Gamma_1 + \Gamma_2 \mid \mathsf{m} \cdot \Gamma
\end{array}
$$

Figure 3.3: Grammar of λ_d

3.5 Type system

λ_d is a simply typed λ-calculus with unit (1), product ($\otimes$) and sum ($\oplus$) types. Its most salient features are the destination $\lfloor_n \mathsf{T} \rfloor$ and ampar $\mathsf{S} \ltimes \mathsf{T}$ types which we've introduced in Sections 3.2 to 3.4. Just as important are *modes* and the associated exponential modality $!_m$. The grammar of λ_d is presented in Figure 3.3. Some of the constructions that we've been using in Sections 3.2 to 3.4 are syntactic sugar for more fundamental forms, we give their definitions in Figure 3.4.

Following a common trend (e.g. [Ber+18; Atk18; OLE19; AB20]), our modes form a semiring[3]. In λ_d the mode semiring is the product of a *multiplicity* semiring for linearity, as in [Ber+18], and of an *age* semiring (see Section 3.5) to prevent the scoping issues discussed in Section 3.3.

We usually omit mode annotations when the mode is the unit element 1_ν. In particular, a function arrow without annotation, or with multiplicity annotation 1, is linear; it is equivalent to the linear arrow $\multimap$ from [Gir95].

[3]Technically, our semirings are commutative but don't have a zero. The terminology "ringoid" has been sometimes used for semiring without neutral elements (neither zero nor unit), for instance [AB20]. So maybe a more accurate terminology would be commutative ringoids with units. We'll stick to "semiring" from now on.

$$() \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{map}\ \mathsf{alloc}\ \mathtt{with}\ d \mapsto d \triangleleft ())$$

$$\mathsf{Inl}\ t \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{map}\ \mathsf{alloc}\ \mathtt{with}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft t)$$

$$\mathsf{Inr}\ t \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{map}\ \mathsf{alloc}\ \mathtt{with}\ d \mapsto d \triangleleft \mathsf{Inr} \blacktriangleleft t)$$

$$\mathsf{Mod}_{\mathsf{m}}\ t \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{map}\ \mathsf{alloc}\ \mathtt{with}\ d \mapsto d \triangleleft \mathsf{Mod}_{\mathsf{m}} \blacktriangleleft t)$$

$$\mathtt{from}'_{\ltimes}\ t \triangleq \mathtt{case}\ (\mathtt{from}_{\ltimes}\ (\mathtt{map}\ t\ \mathtt{with}\ un \mapsto un \fatsemi \mathsf{Mod}_{1_\infty}\ ()))\ \mathtt{of}\ (st\,, ex) \mapsto \mathtt{case}\ ex\ \mathtt{of}\ \mathsf{Mod}_{1_\infty}\ un \mapsto un \fatsemi st$$

$$\lambda x\ {}_{\mathsf{m}}\!\mapsto u \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{map}\ \mathsf{alloc}\ \mathtt{with}\ d \mapsto d \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\mapsto u))$$

$$(t_1\,, t_2) \triangleq \mathtt{from}'_{\ltimes}\,(\mathtt{map}\ \mathsf{alloc}\ \mathtt{with}\ d \mapsto \mathtt{case}\ (d \triangleleft (,))\ \mathtt{of}\ (d_1\,, d_2) \mapsto d_1 \blacktriangleleft t_1 \fatsemi d_2 \blacktriangleleft t_2)$$

Figure 3.4: Syntactic sugar for terms

$+$	$\uparrow^n$	∞
$\uparrow^m$	if $n = m$ then $\uparrow^n$ else ∞	∞
∞	∞	∞

$\cdot$	$\uparrow^n$	∞
$\uparrow^m$	$\uparrow^{n+m}$	∞
∞	∞	∞

$+$	1	ω
1	ω	ω
ω	ω	ω

$\cdot$	1	ω
1	1	∞
ω	∞	∞

$\nu \triangleq \uparrow^0 \qquad \uparrow \triangleq \uparrow^1$

Figure 3.5: Operation tables for age and multiplicity semirings

The age semiring

In order to prevent destinations from escaping their scope, as discussed in Section 3.3, we track the *age* of destinations. Specifically we track, with a de-Bruijn-index-like discipline, what scope a destination originates from. We'll see in Section 3.5 that scopes are introduced by $\mathtt{map}\ t\ \mathtt{with}\ x \mapsto t'$. If we have a term $\mathtt{map}\ t_1\ \mathtt{with}\ x_1 \mapsto \mathtt{map}\ t_2\ \mathtt{with}\ x_2 \mapsto \mathtt{map}\ t_3\ \mathtt{with}\ x_3 \mapsto x_1$, then the innermost occurrence of x_1 has age $\uparrow^2$ because two nested `map` separates the definition and use site of x_1.

We also have an age ∞ for values which don't originate from the scope of a $\mathtt{map}\ t\ \mathtt{with}\ x \mapsto t'$ and can be freely used in and returned by any scope. In particular, destinations can never have age ∞; the main role of age ∞ is thus to act as a guarantee that a value doesn't contain destinations. Finally, we will write $\nu \triangleq \uparrow^0$ for the age of destination that originate from the current scope; and $\uparrow \triangleq \uparrow^1$ as we will frequently multiply ages by $\uparrow$, when entering a scope, to mean that in the scope, all the free variables have their age increased by 1.

I removed the bit about extending ages to variables because we, in fact, have no age (or multiplicity, or mod) variables.

This description is reflected by the semiring operations. Multiplication $\cdot$ is used when nesting a term inside another: then ages, as indices, are summed. Addition $+$ is used to share a variable between two subterms, it ought to be read as giving the variable the same age on both sides. Tables for the $+$ and $\cdot$ operations are presented in Figure 3.5.

Design motivation behind the ampar and destination types

Minamide's work [Min98] is the earliest record we could find of a functional calculus in which incomplete data structures can exists as first class values, and be composed. Crucially, such structures don't have to be completed immediately, and can act as actual containers, e.g. to implement different lists as in Section 3.2.

In [Min98], a structure with a hole is named *hole abstraction*. In the body of a hole abstraction, the bound *hole variable* should be used linearly (exactly once), and must only be used as a parameter of a data constructor (it cannot be pattern-matched on). A hole abstraction of type $(\mathsf{T}, \mathsf{S})\mathtt{hfun}$ is thus a weak form of linear lambda abstraction $\mathsf{T} \multimap \mathsf{S}$, which just moves a piece of data into a bigger data structure.

Now, in classical linear logic (CLL), we know we can transform linear implication $\mathsf{T} \multimap \mathsf{S}$ into $\mathsf{S} \parr \mathsf{T}^\perp$. Doing so for the type $(\mathsf{T}, \mathsf{S})\mathtt{hfun}$ gives $\mathsf{S} \parr \lfloor \mathsf{T} \rfloor$, where $\lfloor \bullet \rfloor$ is a form of dualisation. We use a slightly abusive notation here; this is not exactly the same *par* connective as in CLL, because `hfun` is not as powerful as $\multimap$.

Transforming the hole abstraction from its original implication form to a *par* form let us consider the dualized type $\lfloor \mathtt{T} \rfloor$ — that we call *destination* type — as a first-class component of our calculus. We also get to see the hole abstraction as a pair-like structure (like it is implemented in practice), where the two sides might be coupled together in a way that prevent using both of them simultaneously.

From par ⅋ to ampar ⋉ In CLL, thanks to the cut rule, any of the sides $\mathtt{S}$ or $\mathtt{T}$ of a par $\mathtt{S} \parr \mathtt{T}$ can be eliminated, by interaction with the opposite type $\bullet^\perp$, which then frees up the other side. But in λ_d, we have two types of interaction to consider: interaction between $\mathtt{T}$ and $\lfloor \mathtt{T} \rfloor$, and interaction between $\mathtt{T}$ and $\mathtt{T} \multimap \bullet$. The structure containing holes, $\mathtt{S}$, can safely interact with $\lfloor \mathtt{S} \rfloor$ (merge it into another structure with holes), but not with $\mathtt{T} \multimap \bullet$, as it would let us read an incomplete structure!

On the other hand, a complete value of type $\mathtt{T} = (\dots \lfloor \mathtt{T}' \rfloor \dots)$ containing destinations (but no holes) can safely interact with a function $\mathtt{T} \multimap \mathtt{1}$: in particular, the function can pattern-match on the value of type $\mathtt{T}$ to access destination $\lfloor \mathtt{T}' \rfloor$. However, it might not be safe to fill the $\mathtt{T} = (\dots \lfloor \mathtt{T}' \rfloor \dots)$ into a $\lfloor \mathtt{T} \rfloor$ as that might allow scope escape of the destination $\lfloor \mathtt{T}' \rfloor$ as we've just seen in Section 3.3.

As a result, we cannot adopt rules from CLL blindly. We must be even more cautious since the destination type is not an involutive dualisation, unlike CLL one.

To recover sensible rules for the connective, we decided to make it asymmetric, hence ampar ($\mathtt{S} \ltimes \mathtt{T}$) for *asymmetrical memory par*:

- the left side $\mathtt{S}$ can contain holes, and can be only be eliminated by interaction with $\lfloor \mathtt{S} \rfloor$ using operator $\triangleleft\circ$ to free up the right side $\mathtt{T}$;

- the right side $\mathtt{T}$ cannot contain holes (it might contain destinations), and can be eliminated by interaction with $\mathtt{T} \multimap \mathtt{1}$ to free up the left side $\mathtt{S}$. This is done using $\mathtt{from}'_\ltimes$ and $\mathtt{map}$.

Typing rules

The typing rules for λ_d are highly inspired from[AB20] and Linear Haskell [Ber+18], and are detailed in Figure 3.6. In particular, we use the same additive/multiplicative approach on contexts for linearity and age enforcement. For that we need two operations:

- We lift mode multiplication to typing contexts as a pointwise operation on bindings; we pose $\mathtt{n}' \cdot (x :_{\mathtt{m}} \mathtt{T}) \triangleq x :_{\mathtt{n}' \cdot \mathtt{m}} \mathtt{T}$.

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

Figure 3.6 presents the typing rules for terms, and rules for syntactic sugar forms that have been derived from term rules and proven formally too. Figure 3.9 presents the typing rules for values of the language. We'll now walk through the few peculiarities of the type system for terms.

The predicate $\mathtt{DisposableOnly}\ \Gamma$ in rules Ty-term-Var, Ty-term-Alloc and Ty-sterm-Unit says that Γ can only contain bindings with multiplicity ω, for which weakening is allowed in linear logic. It is enough to allow weakening at the leaves of the typing tree, *i.e.* in the three aforementioned rules.

All the bit about mode ordering should move in the section about modes

Rule Ty-term-Var, in addition to weakening, allows for dereliction of the mode for the variable used, with subtyping constraint $1\nu \leq \mathtt{m}$ defined as $\mathtt{p}\alpha \leq \mathtt{p}'\alpha' \iff \mathtt{p} \leq^{\mathtt{p}} \mathtt{p}' \wedge \alpha \leq^{\mathtt{a}} \alpha'$ where:

$$\begin{cases} 1 \leq^{\mathtt{p}} 1 \\ \mathtt{p} \leq^{\mathtt{p}} \omega \end{cases} \qquad \begin{cases} \uparrow^m \leq^{\mathtt{a}} \uparrow^n \iff m = n \quad \text{(no finite age dereliction ; recall that } \uparrow^0 = \nu\text{)} \\ \alpha \leq^{\mathtt{a}} \infty \end{cases}$$

Rule Ty-term-PatU is elimination for unit, and is also used to chain fill operations.

Pattern-matching with rules Ty-term-App, Ty-term-PatS, Ty-term-PatP and Ty-term-PatE is parametrized by a mode $\mathtt{m}$ by which the typing context Γ_1 of the scrutinee is multiplied. The variables which bind the subcomponents of the scrutinee then inherit this mode. In particular, this choice enforces the equivalence $!_{\omega\alpha}(\mathtt{T}_1 \otimes \mathtt{T}_2) \simeq (!_{\omega\alpha}\mathtt{T}_1) \otimes (!_{\omega\alpha}\mathtt{T}_2)$, which is not part of intuitionistic linear logic, but valid in Linear Haskell [Ber+18].

Rules for scoping As destinations always exist in the context of a structure with holes, and must stay in that context (see Section 3.3), we need a formal notion of *destination scope*. Destination scopes (we'll usually just say *scopes*) are created by rule Ty-term-Map, as destinations are only ever accessed through `map`. More precisely, `map` t `with` $x \mapsto t'$ creates a new scope for x which spans over t'. In that scope, x has age ν ("now"), and the age of the other bindings in the typing context is incremented by 1 (i.e. scaled by $\uparrow$). We see that t' types in $1\uparrow\cdot\Gamma_2,\ x :_{1\nu} \mathsf{T}$ while the global term `map` t `with` $x \mapsto t'$ mentions unscaled context Γ_2. The notion of age, that we attach on bindings, lets us distinguish x — introduced by `map` to bind the right-hand side of the ampar, containing destinations — from anything else that was previously bound, and this information is propagated throughout the typing of t'. Specifically, distinguishing the age of destinations is crucial when typing destination-filling primitives.

Figure 3.7 illustrates scopes introduced by `map`, and how the typing rules for `map` and ◀ interact.

Anticipating Section 3.6, ampar values are pairs with a structure with holes on the left, and destinations on the right. With `map` 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 rule Ty-term-FillLeaf, for instance $x_1 \blacktriangleleft x_0$ in the figure, we type x_1 in the new scope, while we type x_0 in the outer scope, as it's being moved to the structure with holes on the left of the ampar, which lives in the outer scope too. This is, in fact the opposite of the scaling that `map` does: while `map` creates a new scope for its body, operator ◀, and similarly, $\triangleleft\!\circ$ and $\triangleleft(\lambda x\ {}_{\mathtt{m}}\!\mapsto u)$, transfer their right operand to the outer scope. We chose this destination-filling form for function creation because of that similarity, and so that any data can be built through piecemeal destination filling, for consistency.

When using $\mathtt{from}'_{\ltimes}$ (rule Ty-sterm-FromA'), the left of an ampar is extracted to the current scope (as seen at the bottom of Figure 3.7 with x_{22}): this is the fundamental reason why the left of an ampar has to "take place" in the current scope. We know the structure is complete and can be extracted because the right side is of type unit (1), and thus no destination on the right side means no hole can remain on the left. $\mathtt{from}'_{\ltimes}$ is implemented in terms of $\mathtt{from}_{\ltimes}$ in Figure 3.4 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 complete and disposed of with the more general $\mathtt{from}_{\ltimes}$ in rule Ty-term-FromA however, we extract both sides of the ampar to the current scope, even though the right side is normally in a different scope. This is only safe to do because the right side is required to have type $!_{1\infty}\mathsf{T}$, which means it is scope-insensitive: it cannot contain any scope-controlled resource. In particular this ensures that the right side cannot contain destinations, meaning that the structure on the left is complete and ready to be read.

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

The remaining operators $\triangleleft(), \triangleleft\mathtt{Inl}, \triangleleft\mathtt{Inr}, \triangleleft\,\mathtt{Mod}_{\mathtt{m}}, \triangleleft(,)$ from rules Ty-term-Fill$*$ 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 in the process (or unit if there is none).

3.6 Operational semantics

Before we define the operational semantics of λ_d we need to introduce a few more concepts. We'll need commands $E[t]$, they're described in Section 3.6; and we'll need values, described in 3.3. Indeed, the terms of λ_d lack any way to represent destinations or holes, or really any kind of value (for instance $\mathtt{Inl}\,()$ has been, so far, just syntactic sugar for a term $\mathtt{from}'_{\ltimes}\ (\mathtt{map}\ \mathtt{alloc}\ \mathtt{with}\ \ldots)$). It's a peculiarity of λ_d that values only exist during the reduction, in this aspect our operational semantics resembles a denotational semantics. We sometimes call values *runtime values* to emphasize this aspect. In order to express type safety with respect to our operational semantics, we'll need to extend the type system to cover commands and values, but these new typing rules are better thought of as technical device for the proofs than as part of the type system proper.

Runtime values and extended terms

The syntax of runtime values is given in Figure 3.8. It features constructors for all of our basic types, as well as functions (note that in $\lambda x\,{}_{\mathsf{m}} \mapsto u$, u is a term, not a value). The more interesting values are holes $\boxed{h}$, destinations $\to h$, and ampars ${}_{\#}\langle v_2 \,{}_{\wedge} v_1\rangle$, which we'll describe in the rest of the section. In order for the operational semantics to use substitution, which requires substituting variables with values, we also extend the syntax of terms to include values.

Destinations and holes are two faces of the same coin, as seen in Section 3.2, 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 type system is to feature *hole bindings* $\boxed{h} :_{\mathsf{n}} \mathsf{T}$ and *destination bindings* $\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor$ in addition to the variable bindings $x :_{\mathsf{m}} \mathsf{T}$ that usually populates typing contexts. In both cases, we call h a *hole name*. By definition, a context Θ can contain both destination bindings and hole bindings, but *not a destination binding and a hole binding for the same hole name*.

We need to extend our previous context operations to act on the new binding forms:

$$\begin{cases} \mathsf{n}' \cdot (x :_{\mathsf{m}} \mathsf{T}) &= x :_{\mathsf{n}'\cdot\mathsf{m}} \mathsf{T} \\ \mathsf{n}' \cdot (\boxed{h} :_{\mathsf{n}} \mathsf{T}) &= \boxed{h} :_{\mathsf{n}'\cdot\mathsf{n}} \mathsf{T} \\ \mathsf{n}' \cdot (\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor) &= \to h :_{\mathsf{n}'\cdot\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor \end{cases}$$

$$\begin{cases} (x :_{\mathsf{m}} \mathsf{T}) + (x :_{\mathsf{m}'} \mathsf{T}) &= x :_{\mathsf{m}+\mathsf{m}'} \mathsf{T} \\ (\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\boxed{h} :_{\mathsf{n}'} \mathsf{T}) &= \boxed{h} :_{\mathsf{n}+\mathsf{n}'} \mathsf{T} \\ (\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor) + (\to h :_{\mathsf{m}'} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor) &= \to h :_{\mathsf{m}+\mathsf{m}'} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor \quad \textit{(note that } \mathsf{n} \textit{ is the same in both)} \\ (\mathit{name} :_{\mathsf{m}} \mathsf{T}) + \Omega &= \mathit{name} :_{\mathsf{m}} \mathsf{T}, \Omega \quad \textit{if } \mathit{name} \notin \Omega \end{cases}$$

The last definition ranges over $\mathit{name} ::= x \mid \boxed{h} \mid \to h$ and $\Omega ::= \Gamma \mid \Theta \mid \Delta$. Context addition is still very partial; for instance, $(\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}'} \mathsf{T}\rfloor)$ is not defined, as h is present on both sides but with different binding forms.

One of the main goals of λ_d is to ensure that a hole value is never read. The type system (Figure 3.9) maintains this invariant by simply not allowing any hole bindings in the context when typing terms (see Figure 3.8 for the different type of contexts used in the typing judgment). In fact, the only place where holes are introduced, is the left-hand side v_2 in an ampar ${}_{\#}\langle v_2 \,{}_{\wedge} v_1\rangle$, in rule Ty-val-Ampar.

Specifically, holes come from the operator $\to^{-1}$, which represents the matching hole bindings for a set of destination bindings. It is a partial operation on destination-binding contexts defined pointwise as: $\to^{-1}(\to h :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T}\rfloor) = \boxed{h} :_{\mathsf{n}} \mathsf{T}$. Note that $\to^{-1}$ is undefined if any of the bindings has a mode other than 1ν.

Furthermore, in rule Ty-val-Ampar, the holes and the corresponding destination are bound: this is how we ensure that, indeed, there's one destination per hole and one hole per destination.

On the other hand, both sides of the ampar may also contain stored destinations from other scopes, represented by $1\uparrow \cdot \Delta_1$ and Δ_2 in the respective typing contexts of v_1 and v_2.

Not sure whether we should explain why Δ_1 is offset by $1\uparrow$ in the premise but not in conclusion. It's hard to come up with an intuitive narrative.

TODO: Revisit LinOnly / FinAgeOnly ?

The properties $\mathtt{LinOnly}\ \Delta_3$ and $\mathtt{FinAgeOnly}\ \Delta_3$ are true given that $\to^{-1}\Delta_3$ is a valid typing context, so are not really a new restriction on Δ_3. They are mostly used to ease the mechanized proof of type safety for the system.

We haven't defined what LinOnly and FinAgeOnly mean. This is intuitive, but still. Can we simply remove LinOnly and FinOnly from the rule? This way we can remove this paragraph

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

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

One might wonder how we can represent a curried function $\lambda x \mapsto \lambda y \mapsto x$ **concat** y as 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 $\text{`}\lambda x \mapsto \textbf{from}'_{\ltimes}\,(\textbf{map}\ \texttt{alloc}\ \textbf{with}\ d \mapsto d \triangleleft (\lambda y \mapsto x\ \textbf{concat}\ y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\lambda y \mapsto x\ \textbf{concat}\ y)$ is part of term syntax, it's allowed to have free variable x.

TODO: Revisit this paragraph

Rules Ty-val-Hole and Ty-val-Dest are interesting. The mode must be exactly 1_ν: there is no mode weakening for hole or destinations. Typing of value keeps track precisely of holes and destinations during evaluation. Which is what we need to make sure that holes are written before they are read.

Consequently the mode n of a hole binding $\boxed{h} :_{\mathsf{n}} \mathsf{T}$ arises precisely when a structure with holes has exponentials e.g. $\texttt{Mod}_{\omega_\nu}\ \boxed{h}$. It means that only a value with mode n (more precisely, a value typed in a context of the form $\mathsf{n} \cdot \Theta$) can be written to $\boxed{h}$.

Destination bindings $\rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor$, on the other hand, mentions two modes: m and n; m is the mode of the destination as a value: it tracks on the age of $\rightarrow h$. Whereas n is part of the destination's type $\lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ and means that only values with mode n can fill $\rightarrow h$. In a well-typed closed program the m can never be of multiplicity ω or age ∞; a destination is always linear and of finite age.

(Yann) I know it's just a phrasing, but as it is it sounds to naively contradict the intro of the section which says explicitly that modes aren't part of types, which got me a bit surprised in the first read. (Arnaud) I don't think we're saying this anymore

Evaluation contexts and commands

The semantics of λ_d is given using small-step reductions on a pair $E[t]$ of an evaluation context E, and an (extended) term t under focus. We call such a pair $E[t]$ a *command*, borrowing the terminology from [CH00]. We use the notation usually reserved for one-hole contexts because it makes most reduction rules familiar, but it's important to keep in mind that $E[t]$ is formally a pair, which won't always have a clear corresponding term.

The intuition behind our using such commands is that destination actually require a very tame notion of state. So tame, in fact, that we can simply represent writing to a hole by a mere substitution in the evaluation context.

TODO: Fix overflowing figure

The grammar of evaluation contexts is given in Figure 3.10. An evaluation context E is the composition of an arbitrary number of focusing components $e_1, e_2 \ldots$. We chose to represent this composition explicitly using a stack, instead of a meta-operation that would only let us access its final result. As a result, focusing and defocusing operations are made explicit in the semantics, resulting in a more verbose but simpler proof. It is also easier to imagine how to build a stack-based interpreter for such a language.

Focusing components are all directly derived from the term syntax, except for the *open ampar* focusing component $\overset{\text{op}}{H}\langle v_2 \wedge \square \rangle$. This focusing component indicates that an ampar is currently being **map**ped on, 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 **map**'s body and closed back afterwards is the counterpart to the notion of scopes in the typing rules.

We introduce a special substitution $E(\!|h :=_H v|\!)$ that is used to update structures under construction that are attached to open ampar focusing components in the stack. Such a substitution is triggered when a destination $\rightarrow h$ is filled in the term under focus, and results in the value v (that may contain holes itself, e.g. if it is a hollow constructor $(\boxed{h_1}, \boxed{h_2})$) being written to the hole $\boxed{h}$ (that must appear somewhere on an open ampar focusing component). The set H tracks the potential hole names introduced by value v, and is used to update the hole name set of the ampar where h lives:

$$\begin{aligned}
\left(E \circ \overset{\text{op}}{\{h\} \sqcup H}\langle v_2 \wedge \square \rangle\right)(\!|h :=_{H'} v'|\!) &= E \circ \overset{\text{op}}{H \sqcup H'}\langle v_2 (\!|h :=_{H'} v'|\!) \wedge \square \rangle \\
(E \circ e)(\!|h :=_{H'} v'|\!) &= E(\!|h :=_{H'} v'|\!) \circ e \qquad \textit{otherwise}
\end{aligned}$$

Evaluation contexts E are typed in a context Δ that can only contains destination bindings. As we can see in rule Ty-cmd, Δ is exactly the typing context that the term t has to use to form the command $E[t]$.

In other words, while $\Gamma \vdash t : \mathsf{T}$ *requires* the binding in context Γ, $\Delta \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0$ *provides* the bindings in Δ. The typing rules for evaluation contexts E and commands $E[t]$ are presented in Figure 3.11.

TODO: Fix overflowing figure

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

All typing rules for evaluation contexts can be derived systematically from the ones for the corresponding term (except for the rule Ty-ectxs-OpenAmpar that is a truly new form). Let's take the rule Ty-ectxs-PatP as an example:

Ty-term-PatP

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

Ty-ectxs-PatP

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

- the typing context $\mathtt{m} \cdot \Delta_1,\ \Delta_2$ in the premise for E corresponds to $\mathtt{m} \cdot \Gamma_1 + \Gamma_2$ in the conclusion of Ty-term-PatP;
- the typing context $\Delta_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1,\ x_2 :_{\mathtt{m}} \mathtt{T}_2$ in the premise for term u corresponds to the typing context $\Gamma_2,\ x_1 :_{\mathtt{m}} \mathtt{T}_1,\ x_2 :_{\mathtt{m}} \mathtt{T}_2$ for the same term in Ty-term-PatP;
- the typing context Δ_1 in the conclusion for $E \circ \mathbf{case}_{\mathtt{m}}\ \square\ \mathbf{of}\ (x_1 , x_2) \mapsto u$ corresponds to the typing context Γ_1 in the premise for t in Ty-term-PatP (the term t is located where the focus $\square$ is in Ty-ectxs-OpenAmpar).

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, an 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).

Small-step semantics

We equip λ_d with small-step semantics. There are three sorts of semantic rules:

- focus rules, where we focus on a subterm of a term, by pushing a corresponding focusing component on the stack E;
- unfocus rules, where the term under focus is in fact a value, and thus we pop a focusing component from the stack E and transform it back to the corresponding term so that a redex appears (or so that another focus/unfocus rule can be triggered);
- reduction rules, where the actual computation logic takes place.

Here the focus, unfocus, and reduction rules for PatP:

$$E\left[\mathbf{case}_{\mathtt{m}}\ t\ \mathbf{of}\ (x_1 , x_2) \mapsto u\right] \longrightarrow \left(E \circ \mathbf{case}_{\mathtt{m}}\ \square\ \mathbf{of}\ (x_1 , x_2) \mapsto u\right)\left[t\right] \quad \textit{when} \quad \mathtt{NotVal}\ t$$

$$\left(E \circ \mathbf{case}_{\mathtt{m}}\ \square\ \mathbf{of}\ (x_1 , x_2) \mapsto u\right)\left[v\right] \longrightarrow E\left[\mathbf{case}_{\mathtt{m}}\ v\ \mathbf{of}\ (x_1 , x_2) \mapsto u\right]$$

$$E\left[\mathbf{case}_{\mathtt{m}}\ (v_1 , v_2)\ \mathbf{of}\ (x_1 , x_2) \mapsto u\right] \longrightarrow E\left[u[x_1 := v_1][x_2 := v_2]\right]$$

Rules are triggered in a purely deterministic fashion; once a subterm is a value, it cannot be focused on again. As focusing and defocusing rules are entirely mechanical (they are just a matter of pushing and popping a focusing component on the stack), we only present the set of reduction rules for the system in Figure 3.12, but the whole system is included in the annex (see Figures A.1 and A.2).

Reduction rules for function application, pattern-matching, $\mathbf{to}_{\ltimes}$ and $\mathbf{from}_{\ltimes}$ are straightforward.

All reduction rules for destination-filling primitives trigger a memory write on hole h; we model this as a special substitution $E\langle h :=_{\mathcal{H}} v \rangle$ on the evaluation context E. Red-FillU and Red-FillF do not create any new hole; they only write a value to an existing one. On the other hand, rules Red-FillL, Red-FillR, Red-FillE and Red-FillP all write a hollow constructor to the hole h, that is to say a value containing holes itself. 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 closed, but as soon as an ampar is open, it should have globally unique hole names. This restriction is enforced by premise $\mathtt{hnames}(E)$ ## $\mathtt{hnames}(\Delta_3)$ in rule Ty-ectxs-OpenAmpar for the open ampar focusing component that is created during reduction of **map**. Likewise, any hollow constructor written to a hole should also have globally unique hole names. For simplicity's sake, we assume that hole names are natural numbers.

TODO: Fix renaming with new defs for h', h''...

To obtain globally fresh names, in the premises of the corresponding rules, we first pose $h' = \mathtt{max}(\mathtt{hnames}(E) \cup \{h\}) + 1$ to find the next unused name. Then we use either the *shifted set* $H \pm h' \triangleq \{h + h' \mid h \in H\}$ or the *conditional shift operator*:

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

We extend $\bullet[H \pm h']$ to arbitrary values, extended 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 Open-Ampar and Close-Ampar dictate how and when a closed ampar (a value) is converted to an open ampar (a focusing component) and vice-versa, and they make use of the shifting strategy we've just introduced. With Open-Ampar, the hole names bound by the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing component ${}^{\mathrm{op}}_{H \pm h'}\langle v_2[H \pm h'] \wedge \Box \rangle$ while the right-hand side (containing destinations) is substituted in the body of the **map** statement (which becomes the new term under focus). The rule Close-Ampar triggers when the body of a **map** statement has reduced to a value. In that case, we can close the ampar, by popping the focusing component from the stack E and merging back with v_2 to form a closed ampar again.

In rule Red-FillComp, we write the left-hand side v_2 of a closed ampar ${}_H\langle v_2 \wedge v_1 \rangle$ to a hole $\boxed{h}$ that is part of a structure with holes somewhere inside E. This effectively 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, and thus, we need to make them globally unique, as we do when an ampar is opened with **map**. The renaming is carried out by the conditional shift $v_2[H \pm h'']$ and $v_1[H \pm h'']$.

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

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

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

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

3.7 Formal proof of type safety

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

Most of the proof was done by an author with little prior experience with Coq. This goes to show that Coq is reasonably approachable even for non-trivial development. The proof is about 6000 lines long, and contains nearly 350 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.

Binders are the biggest problem. We've largely managed to make the proof to be only about closed terms, to avoid any complication with binders. This worked up until the substitution lemmas, which is the reason why we haven't proved them in Coq yet (that and the fact that it's much easier to be confident in our pen-and-paper proofs for those).

The proofs aren't particularly elegant. For instance, we don't have any abstract formalization of semirings: it was more expedient to brute-force the properties we needed by hand. We've observed up to 232 simultaneous goals, but a computer makes short work of this: it was solved by a single call to the `congruence` tactic. Nevertheless there are a few points of interest:

- We represent context 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 equality is more complex than with a bespoke data type.
- Addition of context is partial since we can only add two binding of the same name if they also have the same type. Instead of representing addition as a binary function to an optional context, we represent addition as a total function to contexts, but we change contexts to have faulty bindings on some names. This simplifies reasoning about properties like commutativity and associativity, at the cost of having well-formedness preconditions in the premises of typing rules as well as some lemmas.

Mostly to simplify equalities, we assumed a few axioms: functional extensionality, classical logic, and indefinite description:

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

Again this isn't particularly elegant, we could have avoided some of these axioms at the price of more complex development. But for the sake of this article, we decided to favor expediency over elegance.

3.8 Implementation of λ_d using in-place memory mutations

The formal language presented in Sections 3.5 and 3.6 is not meant to be implemented as-is.

First, λ_d doesn't have recursion, this would have obscured the 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. $\mathsf{Mod}_{\omega\nu\ \{h\}}\langle \mathsf{0} :: \boxed{h}_{\wedge} \to h\rangle$ (representing a non-linear difference list $0::\square$), and then used twice, each time in a different way:

$$
\begin{array}{l}
\mathbf{case}\ \mathsf{Mod}_{\omega\nu\ \{h\}}\langle \mathsf{0} :: \boxed{h}_{\wedge} \to h\rangle\ \mathbf{of}\ \mathsf{Mod}_{\omega\nu}\ x \mapsto \\
\quad \mathbf{let}\ x_1 := x\ \mathbf{append}\ 1\ \mathbf{in} \\
\quad \mathbf{let}\ x_2 := x\ \mathbf{append}\ 2\ \mathbf{in} \\
\quad\quad \mathbf{to}_{\mathsf{List}}\ (x_1\ \mathbf{concat}\ x_2) \\
\longrightarrow^*\ \mathsf{0} :: 1 :: \mathsf{0} :: 2 :: []
\end{array}
$$

It may seem counter-intuitive at first, but this program is valid and safe in λ_d. Thanks to the renaming discipline we detailed in Section 3.6, every time an ampar is **mapped** over, its hole names are renamed to fresh ones. One way we can support this is to allocate a fresh copy of x every time we call **append** (which is implemented in terms of **map**), 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 3.2. Copy-on-write can be optimized using fully-in-place functional programming in the style of [LLS23], where, using reference counting, we don't need to perform a copy when the difference list isn't aliased.

An alternative is to refine the linear type system further in order to guarantee that ampars are unique and avoid copy-on-write altogether. For that we follow a recipe proposed by [Spi+22] and introduce a new type `Token`, together with primitives **dup** and **drop** (remember that unqualified arrows have mode 1_ν, so are linear):

$$\begin{aligned}
&\textbf{dup} \ : \ \texttt{Token} \multimap \texttt{Token} \otimes \texttt{Token} \\
&\textbf{drop} \ : \ \texttt{Token} \multimap 1 \\
&\textbf{alloc}_{\texttt{ip}} \ : \ \texttt{Token} \multimap \texttt{T} \ltimes \lfloor \texttt{T} \rfloor \\
&\texttt{alloc}_{\texttt{cow}} \ : \ \texttt{T} \ltimes \lfloor \texttt{T} \rfloor
\end{aligned}$$

We now have two possible allocation primitives: the new one with an in-place mutation memory model (**ip**), that has to be managed linearly, and the old one that doesn't have to used linearly, and features a copy-on-write (`cow`) memory model.

Ampar produced by $\textbf{alloc}_{\texttt{ip}}$ have a linear dependency on a linear tok_k variable. If an ampar produced by $\textbf{alloc}_{\texttt{ip}}\ tok_k$ were to be used twice in a block t, then t would require a typing context $\{tok_k :_{\omega_\nu} \texttt{Token}\}$, that itself would require tok_0 to have multiplicity ω too. Thus the program would be rejected.

Having closed programs to typecheck in non-empty context $\{tok_0 :_{1_\infty} \texttt{Token}\}$ is a slightly more ergonomic solution than adding a primitive function $\textbf{withToken} : (\texttt{Token} \multimap_{1_\infty} !_{\omega_\infty} \texttt{T}) \multimap !_{\omega_\infty} \texttt{T}$ as it is done in [Bag24].

In λ_d with `Token`s and $\textbf{alloc}_{\texttt{ip}}$, as ampars are managed linearly, we can change the allocation and renaming mechanisms:

- the hole name for a new ampar is chosen fresh right from the start (this corresponds to a new heap allocation);
- adding a new hollow constructor still require freshness for its hole names (this corresponds to a new heap allocation too);
- **map**ping over an ampar and filling destinations or composing two ampars using $\triangleleft_\circ$ no longer require any renaming: we have the guarantee that the all the names involved are globally fresh, and can only be used once, so we can do in-place memory updates.

We decided to omit the linearity aspect of ampars in λ_d as it obfuscates the presentation of the system without adding much to the understanding of the latter.

3.9 Related work

Subsection for each work seems a little heavy-weight. On the other hand the italic paragraphs seem too lightweight. I'm longing for the days of paragraph heading in bold.

Destination-passing style for efficient memory management

In [Sha+17], the authors present a destination-based intermediate language for a functional array programming language. They develop a system of destination-specific optimizations and boast near-C performance.

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

(Yann) This could be nuanced: allocating deeply nested trees like ASTs with a bump allocator so that they are packed in memory can also be beneficial, even if they are "linked" data structures, because of cache-friendlyness,

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 consider that these two aspects complement each other: good compiler optimization are important to alleviate the burden from the programmer and allowing high-level abstraction; having the possibility to use destinations in code affords the programmer more control would they need it.

Tail modulo constructor

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

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

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

to ask the compiler to perform the translation. The compiler will then throw an error if it can't. This way, contrary to the optimizations in [Sha+17], this optimization 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 and automatic transformation. On the flip-side, tail modulo constructor is too weak to handle our difference lists or breadth-first traversal examples.

A functional representation of data structures with a hole

The idea of using linear types to let the user manipulate structures with holes safely dates back to [Min98]. Our system is strongly inspired by theirs. In their system, we can only compose functions that represent data structures with holes, but we can't pattern-match on the result; just like in our system we cannot act on the left-hand side of $\mathtt{S} \ltimes \mathtt{T}$, only the right hand part.

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

The more important difference is that while their system is based on a type of linear functions, our is based on the linear logic's par combinator. This, in turns, lets us define a type of destinations which are representations of holes in values, which [Min98] doesn't have. This means that using [Min98] — or the more recent but similarly expressive system from [Lor+24b] — one can implement the examples with difference lists and queues from Section 3.2, but they can't do our breadth-first traversal example from Section 3.4, since storing destinations in a data structure is the essential ingredient of this example.

This ability to store destination does come at a cost though: the system needs this additional notion of ages to ensure that destinations are use soundly. On the other hand, our system is strictly more general, in that the system from [Min98] can be embedded in λ_d, and if one stays in this fragment, we're never confronted with ages. Ages only show up when writing programs that go beyond Minamide's system.

Destination-passing style programming: a Haskell implementation

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

In their system, $d \blacktriangleleft t$ requires t to be unrestricted, while in λ_d, t can be linear. One consequence is that in [Bag24], destinations can be stored in data structures but not in data structures with holes; so in order to do a breadth-first search algorithm like in Section 3.4, they can't use improved queues like we do. The other consequence is that their system require both primitive constructors and destination-filling primitives; it cannot be bootstrapped with the later alone, as we do in the current article.

However, [Bag24] is implemented in Haskell, which just features linear types. Our system subsumes theirs; but it requires the age system that is more than what Haskell provides. Encoding their system in ours will unfortunately make ages appear in the typing rules.

Semi-axiomatic sequent calculus

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

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

3.10 Conclusion and future work

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

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

In fact, we plan to use this very strategy to design an API for destination passing in Haskell, leveraging only the existing linear types, but retaining the possibility of storing destinations in data structures with holes.

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

Ty-term-Var
$$\frac{\mathtt{DisposableOnly}\ \Gamma \qquad 1\nu \leq \mathsf{m}}{\Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}}$$

Ty-term-App
$$\frac{\Gamma_1 \vdash t : \mathsf{T} \qquad \Gamma_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}}$$

Ty-term-PatU
$$\frac{\Gamma_1 \vdash t : \mathsf{1} \qquad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}$$

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

Ty-term-PatP
$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathtt{case}_{\mathsf{m}}\ t\ \mathtt{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}$$

Ty-term-PatE
$$\frac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}}\mathsf{T} \\ \Gamma_2,\ x :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m}\cdot\Gamma_1 + \Gamma_2 \vdash \mathtt{case}_{\mathsf{m}}\ t\ \mathtt{of}\ \mathtt{Mod}_{\mathsf{n}}\ x \mapsto u : \mathsf{U}}$$

Ty-term-Map
$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{U} \ltimes \mathsf{T} \\ 1\uparrow\cdot\Gamma_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathtt{map}\ t\ \mathtt{with}\ x \mapsto t' : \mathsf{U} \ltimes \mathsf{T}'}$$

Ty-term-ToA
$$\frac{\Gamma \vdash u : \mathsf{U}}{\Gamma \vdash \mathtt{to}_{\ltimes}\ u : \mathsf{U} \ltimes \mathsf{1}}$$

Ty-term-FromA
$$\frac{\Gamma \vdash t : \mathsf{U} \ltimes (!_{1\infty}\mathsf{T})}{\Gamma \vdash \mathtt{from}_{\ltimes}\ t : \mathsf{U} \otimes (!_{1\infty}\mathsf{T})}$$

Ty-term-Alloc
$$\frac{\mathtt{DisposableOnly}\ \Gamma}{\Gamma \vdash \mathtt{alloc} : \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor}$$

Ty-term-FillU
$$\frac{\Gamma \vdash t : \lfloor \mathsf{1} \rfloor}{\Gamma \vdash t \triangleleft () : \mathsf{1}}$$

Ty-term-FillL
$$\frac{\Gamma \vdash t : \lfloor \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathtt{Inl} : \lfloor \mathsf{T}_1 \rfloor}$$

Ty-term-FillR
$$\frac{\Gamma \vdash t : \lfloor \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathtt{Inr} : \lfloor \mathsf{T}_2 \rfloor}$$

Ty-term-FillP
$$\frac{\Gamma \vdash t : \lfloor \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor \mathsf{T}_1 \rfloor \otimes \lfloor \mathsf{T}_2 \rfloor}$$

Ty-term-FillE
$$\frac{\Gamma \vdash t : \lfloor !_{\mathsf{n}'}\mathsf{T} \rfloor}{\Gamma \vdash t \triangleleft \mathtt{Mod}_{\mathsf{n}'} : \lfloor \mathsf{n}'\cdot\mathsf{n}\,\mathsf{T} \rfloor}$$

Ty-term-FillF
$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor \mathsf{T} \multimap_{\mathsf{n}} \mathsf{U} \rfloor \\ \Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + (1\uparrow\cdot\mathsf{n})\cdot\Gamma_2 \vdash t \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\mapsto u) : \mathsf{1}}$$

Ty-term-FillComp
$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor \mathsf{U} \rfloor \\ \Gamma_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}\end{array}}{\Gamma_1 + 1\uparrow\cdot\Gamma_2 \vdash t \triangleleft\!\circ\, t' : \mathsf{T}}$$

Ty-term-FillLeaf
$$\frac{\Gamma_1 \vdash t : \lfloor \mathsf{T} \rfloor \qquad \Gamma_2 \vdash t' : \mathsf{T}}{\Gamma_1 + (1\uparrow\cdot\mathsf{n})\cdot\Gamma_2 \vdash t \blacktriangleleft t' : \mathsf{1}}$$

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

Ty-sterm-FromA'
$$\frac{\Gamma \vdash t : \mathsf{T} \ltimes \mathsf{1}}{\Gamma \vdash \mathtt{from}'_{\ltimes}\ t : \mathsf{T}}$$

Ty-sterm-Unit
$$\frac{\mathtt{DisposableOnly}\ \Gamma}{\Gamma \vdash () : \mathsf{1}}$$

Ty-sterm-Fun
$$\frac{\Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_2 \vdash \lambda x\ {}_{\mathsf{m}}\!\mapsto u : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}$$

Ty-sterm-Left
$$\frac{\Gamma_2 \vdash t : \mathsf{T}_1}{\Gamma_2 \vdash \mathtt{Inl}\ t : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-sterm-Right
$$\frac{\Gamma_2 \vdash t : \mathsf{T}_2}{\Gamma_2 \vdash \mathtt{Inr}\ t : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-sterm-Exp
$$\frac{\Gamma_2 \vdash t : \mathsf{T}}{\mathsf{m}\cdot\Gamma_2 \vdash \mathtt{Mod}_{\mathsf{m}}\ t : !_{\mathsf{m}}\mathsf{T}}$$

Ty-sterm-Prod
$$\frac{\Gamma_{21} \vdash t_1 : \mathsf{T}_1 \qquad \Gamma_{22} \vdash t_2 : \mathsf{T}_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1\ ,\ t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}$$

Figure 3.6: Typing rules for terms and syntactic sugar

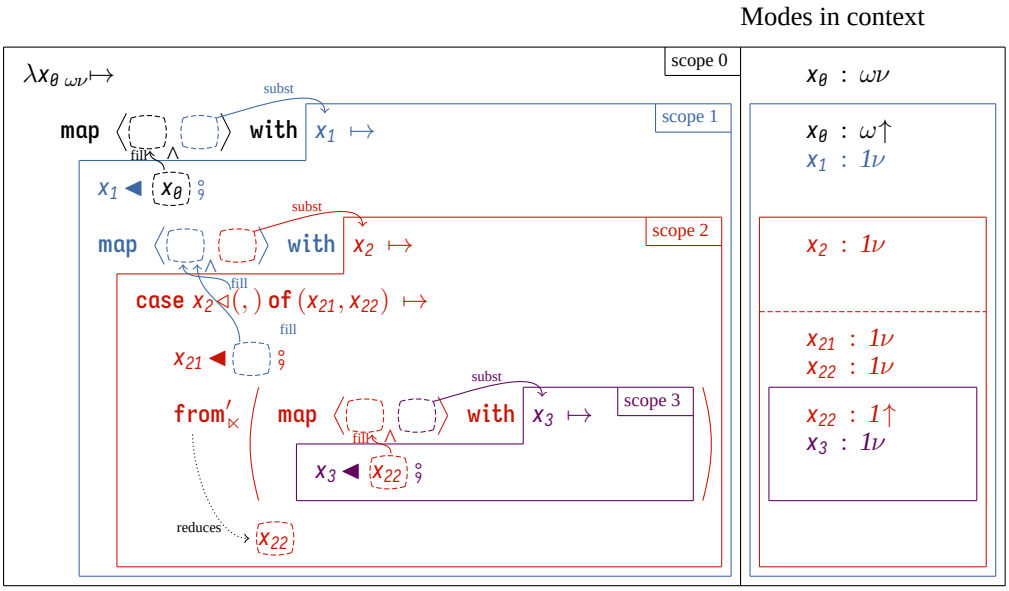


Figure 3.7: Scope rules for `map` in λ_d

$$
\begin{array}{rcll}
t, u & ::= & \ldots \mid v & \\
\\
v & ::= & \boxed{h} & \textit{(hole)} \\
& \mid & \to h & \textit{(destination)} \\
& \mid & {}_H\langle v_2 \wedge v_1 \rangle & \textit{(ampar value form)} \\
& \mid & () \mid \lambda x \,{}_m\!\mapsto u \mid \mathtt{Inl}\, v \mid \mathtt{Inr}\, v \mid \mathtt{Mod}_m\, v \mid (v_1, v_2) &
\end{array}
$$

$$
\begin{array}{rcllllll}
\Delta & ::= & \cdot \mid \to h :_m \lfloor {}_n\mathsf{T} \rfloor & & \mid \Delta_1, \Delta_2 & \mid \Delta_1 + \Delta_2 & \mid m \cdot \Delta \\
\Gamma & ::= & \cdot \mid \to h :_m \lfloor {}_n\mathsf{T} \rfloor & \mid x :_m \mathsf{T} & \mid \Gamma_1, \Gamma_2 & \mid \Gamma_1 + \Gamma_2 & \mid m \cdot \Gamma \\
\Theta & ::= & \cdot \mid \to h :_m \lfloor {}_n\mathsf{T} \rfloor & \mid \boxed{h} :_n \mathsf{T} \mid \to^{-1} \Delta & \mid \Theta_1, \Theta_2 & \mid \Theta_1 + \Theta_2 & \mid m \cdot \Theta
\end{array}
$$

Figure 3.8: Extended terms and runtime values

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(Extended terms)*

Ty-term-Val
$$\frac{\mathtt{DisposableOnly}\ \Gamma \qquad \Delta \vdash v : \mathsf{T}}{\Gamma,\ \Delta \vdash v : \mathsf{T}}$$

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

Ty-val-Hole
$$\frac{}{\boxed{h} :_{1\nu} \mathsf{T} \vdash \boxed{h} : \mathsf{T}}$$

Ty-val-Dest
$$\frac{1\nu \leq \mathsf{m}}{\rightarrow h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor \vdash \rightarrow h : \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor}$$

Ty-val-Unit
$$\frac{}{\cdot \vdash () : \mathbf{1}}$$

Ty-val-Fun
$$\frac{\Delta,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta \vdash \lambda x_{\mathsf{m}} \mapsto u : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}$$

Ty-val-Left
$$\frac{\Theta \vdash v_1 : \mathsf{T}_1}{\Theta \vdash \mathtt{Inl}\, v_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-val-Right
$$\frac{\Theta \vdash v_2 : \mathsf{T}_2}{\Theta \vdash \mathtt{Inr}\, v_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

Ty-val-Prod
$$\frac{\Theta_1 \vdash v_1 : \mathsf{T}_1 \qquad \Theta_2 \vdash v_2 : \mathsf{T}_2}{\Theta_1 + \Theta_2 \vdash (v_1\,,\, v_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}$$

Ty-val-Exp
$$\frac{\Theta \vdash v' : \mathsf{T}}{\mathsf{n} \cdot \Theta \vdash \mathtt{Mod}_{\mathsf{n}}\, v' : !_{\mathsf{n}} \mathsf{T}}$$

Ty-val-Ampar
$$\frac{1\uparrow \cdot \Delta_1,\ \Delta_3 \vdash v_1 : \mathsf{T} \qquad \Delta_2,\ \rightarrow^{-1} \Delta_3 \vdash v_2 : \mathsf{U}}{\Delta_1,\ \Delta_2 \vdash {}_{\mathtt{hnames}(\Delta_3)}\langle v_2 \wedge v_1 \rangle : \mathsf{U} \ltimes \mathsf{T}}$$

Figure 3.9: Typing rules for extended terms and runtime values

$$
\begin{aligned}
E \quad ::= \quad & t'\ \square \ \mid\ \square\ v \ \mid\ \square \fatsemi u \\
\mid\ & \mathtt{case}_{\mathsf{m}}\ \square\ \mathtt{of}\ \{\mathtt{Inl}\, x_1 \mapsto u_1\,,\ \mathtt{Inr}\, x_2 \mapsto u_2\} \ \mid\ \mathtt{case}_{\mathsf{m}}\ \square\ \mathtt{of}\ (x_1\,,\, x_2) \mapsto u \ \mid\ \mathtt{case}_{\mathsf{m}}\ \square\ \mathtt{of}\ \mathtt{Mod}_{\mathsf{n}}\, x \mapsto u \\
\mid\ & \mathtt{map}\ \square\ \mathtt{with}\ x \mapsto t' \ \mid\ \mathtt{to}_{\ltimes}\ \square \ \mid\ \mathtt{from}_{\ltimes}\ \square \\
\mid\ & \square \triangleleft () \ \mid\ \square \triangleleft \mathtt{Inl} \ \mid\ \square \triangleleft \mathtt{Inr} \ \mid\ \square \triangleleft (,) \ \mid\ \square \triangleleft \mathtt{Mod}_{\mathsf{m}} \ \mid\ \square \triangleleft (\lambda x_{\mathsf{m}} \mapsto u) \ \mid\ \square \triangleleft\!\circ\ t' \ \mid\ v \triangleleft\!\circ\ \square \ \mid\ \square \blacktriangleleft t' \ \mid\ v \blacktriangleleft \square \\
\mid\ & {}^{\mathrm{op}}_{H}\langle v_2 \wedge \square \rangle \qquad \textit{(open ampar focusing component)}
\end{aligned}
$$

$$E \quad ::= \quad \square \ \mid\ E \circ E \ \mid\ E\,(\!|h :=_H v|\!)$$

Figure 3.10: Grammar for evaluation contexts

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

Ty-ectxs-Id
$$\frac{}{\bullet \dashv \square : \mathsf{U}_\theta \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-App1
$$\frac{\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\Delta_1 \dashv E \circ t'\ \square : \mathsf{T} \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-App2
$$\frac{\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_1 \vdash v : \mathsf{T}}{\Delta_2 \dashv E \circ \square\ v : (\mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}) \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-PatU
$$\frac{\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ \square \fatsemi u : \mathsf{1} \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-PatS
$$\frac{\begin{array}{c}\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_\theta \\ \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Delta_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\ \square\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : (\mathsf{T}_1 \oplus \mathsf{T}_2) \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-PatP
$$\frac{\begin{array}{c}\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_\theta \\ \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\ \square\ \mathsf{of}\ (x_1\,, x_2) \mapsto u : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-PatE
$$\frac{\begin{array}{c}\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_\theta \\ \Delta_2,\ x :_{\mathsf{m} \cdot \mathsf{m}'} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\ \square\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{m}'}\ x \mapsto u : !_{\mathsf{m}'} \mathsf{T} \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-Map
$$\frac{\begin{array}{c}\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \ltimes \mathsf{T}' \rightarrowtail \mathsf{U}_\theta \\ 1\uparrow \cdot \Delta_2,\ x :_{1_\nu} \mathsf{T} \vdash t' : \mathsf{T}'\end{array}}{\Delta_1 \dashv E \circ \mathsf{map}\ \square\ \mathsf{with}\ x \mapsto t' : (\mathsf{U} \ltimes \mathsf{T}) \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-ToA
$$\frac{\Delta \dashv E : (\mathsf{U} \ltimes \mathsf{1}) \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \mathsf{to}_{\ltimes}\ \square : \mathsf{U} \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FromA
$$\frac{\Delta \dashv E : (\mathsf{U} \otimes (!_{1_\infty} \mathsf{T})) \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \mathsf{from}_{\ltimes}\ \square : (\mathsf{U} \ltimes (!_{1_\infty} \mathsf{T})) \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillU
$$\frac{\Delta \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \square \triangleleft () : \lfloor_{\mathsf{n}} \mathsf{1} \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillL
$$\frac{\Delta \dashv E : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \square \triangleleft \mathsf{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillR
$$\frac{\Delta \dashv E : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \square \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillP
$$\frac{\Delta \dashv E : (\lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor) \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \square \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillE
$$\frac{\Delta \dashv E : \lfloor_{\mathsf{m} \cdot \mathsf{n}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_\theta}{\Delta \dashv E \circ \square \triangleleft \mathsf{Mod}_{\mathsf{m}} : \lfloor_{\mathsf{n}} !_{\mathsf{m}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillF
$$\frac{\begin{array}{c}\Delta_1,\ (1\uparrow \cdot \mathsf{n}) \cdot \Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_\theta \\ \Delta_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Delta_1 \dashv E \circ \square \triangleleft (\lambda x :_{\mathsf{m}} \mapsto u) : \lfloor_{\mathsf{n}} \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillComp1
$$\frac{\Delta_1,\ 1\uparrow \cdot \Delta_2 \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}}{\Delta_1 \dashv E \circ \square \triangleleft_{\circ} t' : \lfloor \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillComp2
$$\frac{\Delta_1,\ 1\uparrow \cdot \Delta_2 \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_1 \vdash v : \lfloor \mathsf{U} \rfloor}{\Delta_2 \dashv E \circ v \triangleleft_{\circ} \square : \mathsf{U} \ltimes \mathsf{T} \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillLeaf1
$$\frac{\Delta_1,\ (1\uparrow \cdot \mathsf{n}) \cdot \Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_2 \vdash t' : \mathsf{T}}{\Delta_1 \dashv E \circ \square \blacktriangleleft t' : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-FillLeaf2
$$\frac{\Delta_1,\ (1\uparrow \cdot \mathsf{n}) \cdot \Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_\theta \qquad \Delta_1 \vdash v : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor}{\Delta_2 \dashv E \circ v \blacktriangleleft \square : \mathsf{T} \rightarrowtail \mathsf{U}_\theta}$$

Ty-ectxs-OpenAmpar
$$\frac{\begin{array}{c}\mathsf{hnames}(E)\ \#\#\ \mathsf{hnames}(\Delta_3) \\ \Delta_1,\ \Delta_2 \dashv E : (\mathsf{U} \ltimes \mathsf{T}') \rightarrowtail \mathsf{U}_\theta \\ \Delta_2,\ \rightarrow^{-1} \Delta_3 \nvdash v_2 : \mathsf{U}\end{array}}{1\uparrow \cdot \Delta_1,\ \Delta_3 \dashv E \circ {}^{\mathrm{op}}_{\mathsf{hnames}(\Delta_3)}\langle v_2 \wedge \square \rangle : \mathsf{T}' \rightarrowtail \mathsf{U}_\theta}$$

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

Ty-cmd
$$\frac{\Delta \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_\theta \qquad \Delta \vdash t : \mathsf{T}}{\vdash E[t] : \mathsf{U}_\theta}$$

Figure 3.11: Typing rules for evaluation contexts and commands

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

$E\,[\,(\lambda x_{\,m} \mapsto u)\ v\,] \longrightarrow E\,[\,u[x := v]\,]$ Red-App

$E\,[\,() \fatsemi u\,] \longrightarrow E\,[\,u\,]$ Red-PatU

$E\,[\,\mathsf{case}_m\ (\mathsf{Inl}\,v_1)\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_1[x_1 := v_1]\,]$ Red-PatL

$E\,[\,\mathsf{case}_m\ (\mathsf{Inr}\,v_2)\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_2[x_2 := v_2]\,]$ Red-PatR

$E\,[\,\mathsf{case}_m\ (v_1\,,\,v_2)\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u\,] \longrightarrow E\,[\,u[x_1 := v_1][x_2 := v_2]\,]$ Red-PatP

$E\,[\,\mathsf{case}_m\ \mathsf{Mod}_n\,v'\ \mathsf{of}\ \mathsf{Mod}_n\,x \mapsto u\,] \longrightarrow E\,[\,u[x := v']\,]$ Red-PatE

$E\,[\,\mathsf{to}_{\ltimes}\,v_2\,] \longrightarrow E\,[\,\{\,\}\langle v_{2\,\wedge}()\rangle\,]$ Red-ToA

$E\,[\,\mathsf{from}_{\ltimes}\ \{\,\}\langle v_{2\,\wedge}\mathsf{Mod}_{1\infty}\,v_1\rangle\,] \longrightarrow E\,[\,(v_2\,,\,\mathsf{Mod}_{1\infty}\,v_1)\,]$ Red-FromA

$E\,[\,\mathsf{alloc}\,] \longrightarrow E\,[\,\{1\}\langle\boxed{1}_{\,\wedge}{\to}1\rangle\,]$ Red-Alloc

$E\,[\,{\to}h \triangleleft ()\,] \longrightarrow E\,(\!|h :=_{\{\,\}}\ ()|\!)\,[\,()\,]$ Red-FillU

$E\,[\,{\to}h \triangleleft (\lambda x_{\,m} \mapsto u)\,] \longrightarrow E\,(\!|h :=_{\{\,\}}\ \lambda x_{\,m} \mapsto u|\!)\,[\,()\,]$ Red-FillF

$E\,[\,{\to}h \triangleleft \mathsf{Inl}\,] \longrightarrow E\,(\!|h :=_{\{h'+1\}}\ \mathsf{Inl}\,\boxed{h'+1}|\!)\,[\,{\to}h'+1\,]$ Red-FillL

$E\,[\,{\to}h \triangleleft \mathsf{Inr}\,] \longrightarrow E\,(\!|h :=_{\{h'+1\}}\ \mathsf{Inr}\,\boxed{h'+1}|\!)\,[\,{\to}h'+1\,]$ Red-FillR

$E\,[\,{\to}h \triangleleft \mathsf{Mod}_m\,] \longrightarrow E\,(\!|h :=_{\{h'+1\}}\ \mathsf{Mod}_m\,\boxed{h'+1}|\!)\,[\,{\to}h'+1\,]$ Red-FillE

$E\,[\,{\to}h \triangleleft (,)\,] \longrightarrow E\,(\!|h :=_{\{h'+1,h'+2\}}\ (\boxed{h'+1}\,,\,\boxed{h'+2})|\!)\,[\,({\to}h'+1\,,\,{\to}h'+2)\,]$ Red-FillP

$E\,[\,{\to}h \triangleleft\!\circ_H \langle v_{2\,\wedge}v_1\rangle\,] \longrightarrow E\,(\!|h :=_{(H \pm h'')}\ v_2[H \pm h'']|\!)\,[\,v_1[H \pm h'']\,]$ Red-FillComp

$E\,[\,{\to}h \blacktriangleleft v\,] \longrightarrow E\,(\!|h :=_{\{\,\}}\ v|\!)\,[\,()\,]$ Red-FillLeaf

$E\,[\,\mathsf{map}_H\langle v_{2\,\wedge}v_1\rangle\ \mathsf{with}\ x \mapsto t'\,] \longrightarrow \left(E \circ {}^{\mathrm{op}}_{H \pm h'''}\langle v_2[H \pm h''']_{\,\wedge}\Box\rangle\right)\,[\,t'[x := v_1[H \pm h''']]\,]$ Open-Ampar

$\left(E \circ {}^{\mathrm{op}}_{H}\langle v_{2\,\wedge}\Box\rangle\right)\,[\,v_1\,] \longrightarrow E\,[\,{}_H\langle v_{2\,\wedge}v_1\rangle\,]$ Close-Ampar

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

Figure 3.12: Small-step semantics

Chapter 4

Destination-passing style : a Haskell implementation

4.1 Introduction

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

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

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

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

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

The main contributions of this paper are

- a linearly-typed API for destinations that let us build and manipulate data structures with holes while exposing a pure interface (Section 4.4);
- a first implementation of destinations for Haskell relying on so-called compact regions (Section 4.5), together with a performance evaluation (Section 4.6). Implementation code is available in [Bag23a] and [Bag23b] (specifically `src/Compact/Pure/Internal.hs` and `bench/Bench`).

4.2 A short primer on linear types

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

A value is said to be *consumed once* (or *consumed linearly*) when it is pattern-matched on and its sub-components are consumed once; or when it is passed as an argument to a linear function whose result is consumed once. A function is said to be *consumed once* when it is applied to an argument and when the result is consumed exactly once. We say that a variable `x` is *used linearly* in an expression `u` when consuming `u` once implies consuming `x` exactly once. Linearity on function arrows thus creates a chain of requirements about consumption of values, which is usually bootstrapped by using the *scope function* trick, as detailed in Section 4.4.

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

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

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

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

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

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

4.3 Motivating examples for DPS programming

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

Efficient difference lists

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

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

```
1  data [a] = {- nil constructor -} [] | {- cons constructor -} (:) a [a]
2
3  type DList a = Incomplete [a] (Dest [a])
4
5  alloc :: DList a  -- API primitive (simplified signature w.r.t. Section 4)
6
7  append :: DList a ⊸ a → DList a
8  append i x =
9    i <&> \d → case fill @'(:) d of
10     (dh, dt) → fillLeaf x dh ; dt
11
12 concat :: DList a ⊸ DList a ⊸ DList a
13 concat i1 i2 = i1 <&> \dt1 → fillComp i2 dt1
14
15 toList :: DList a ⊸ [a]
16 toList i = fromIncomplete_' (i <&> \dt → fill @'[] dt)
```

Listing 1: Implementation of difference lists with destinations

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

With such a representation, concatenation is function composition: `f1 <> f2 = f1 . f2`, and we have `mempty = id`[1], `toList f = f []` and `fromList xs = \ys → xs ++ ys`.

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

With the function encoding, it isn't possible to read the list until a last element of type `[a]` has been supplied to complete it. With the destination representation, this constraint must persist: the actual list `[a]` shouldn't be readable until the accompanying destination is filled, as pattern-matching on the hole would lead to a dreaded *segmentation fault*. This constraint is embodied by the `Incomplete a b` type of our destination API: `b` is what needs to be linearly consumed to make the `a` readable. The `b` side often carries the destinations of a structure. A difference list is then `type DList a = Incomplete [a] (Dest [a])`: the `Dest [a]` must be filled (with a `[a]`) to get a readable `[a]`.

The implementation of destination-backed difference lists is presented in Listing 1. More details about the API primitives used by this implementation are given in Section 4.4. For now, it's important to note that `fill` is a function taking a constructor as a type parameter (often used with `@` for type parameter application, and `'` to lift a constructor to a type).

- `alloc` (Figure 4.1) returns a `DList a` which is exactly an `Incomplete [a] (Dest [a])` structure. There is no data there yet and the list that will be fed in `Dest [a]` is exactly the list that the resulting `Incomplete` will hold. This is similar to the function encoding where `\x → x` represents the empty difference list;
- `append` (Figure 4.3) adds an element at the tail position of a difference list. For this, it first uses `fill @'(:)` to fill the hole at the end of the list represented by `d :: Dest [a]` with a hollow *cons* cell with two new holes pointed by `dh :: Dest a` and `dt :: Dest [a]`. Then, `fillLeaf` fills the hole represented by `dh` with the value of type `a` to append. The hole of the resulting difference list is the one pointed by `dt :: Dest [a]` which hasn't been filled yet.

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

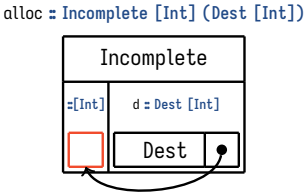


Figure 4.1: `alloc`

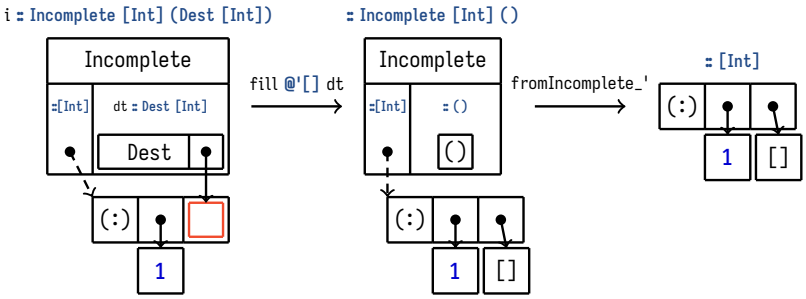


Figure 4.2: Memory behavior of `toList i`

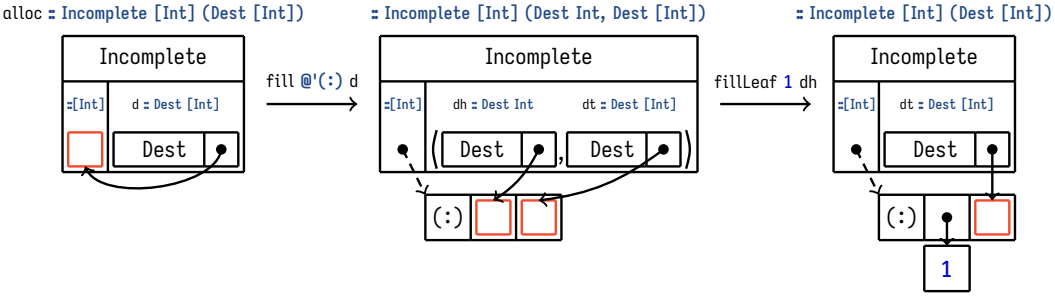


Figure 4.3: Memory behavior of `append alloc 1`

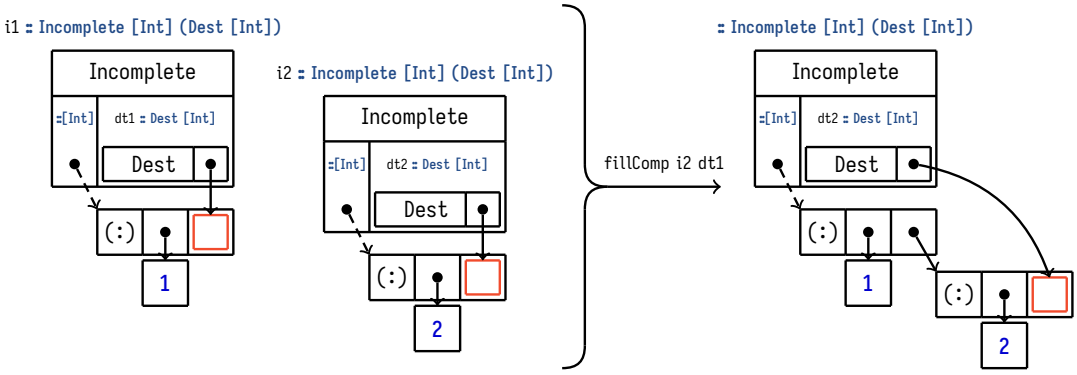


Figure 4.4: Memory behavior of `concat i1 i2` (based on `fillComp`)

- `concat` (Figure 4.4) concatenates two difference lists, `i1` and `i2`. It uses `fillComp` to fill the destination `dt1` of the first difference list with the root of the second difference list `i2`. The resulting `Incomplete` object hence has the same root as the first list, holds the elements of both lists, and inherits the hole of the second list. Memory-wise, `concat` just writes an address into a memory cell; no move is required.
- `toList` (Figure 4.2) completes the incomplete structure by plugging *nil* into its hole with `fill @'[]` (whose individual behavior is presented in Figure 4.6) and removes the `Incomplete` wrapper as the structure is now complete, using `fromIncomplete_'`.

To use this API safely, it is imperative that values of type `Incomplete` are used linearly. Otherwise we could first complete a difference list with `l = toList i`, then add a new cons cell with a hole to `i` with `append i x` (actually reusing the destination inside `i` for the second time). Doing that creates a hole inside `l`, although it is of type `[a]` so we are allowed to pattern-match on it (so we might get a segfault)! The simplified API of Listing 1 doesn't actually enforce the required linearity properties, I'll address that in Section 4.4.

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

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

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

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

This implementation of difference list matches closely the intended memory behaviour, we can expect it to be more efficient than the functional encoding. We'll see in Section 4.6 that the prototype implementation presented in Section 4.5 cannot yet demonstrate these performance improvements.

Breadth-first tree traversal

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

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

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

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

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

As the state-transforming function `s → a → (s, b)` is non-linear, the nodes of the original tree won't be used in a linear fashion. However, we want to store these nodes together with their accompanying destinations in a queue of pairs, and destinations used to construct the queue have to be used linearly (because of `<&>` signature, which initially gives the root destination). That forces the queue to be used linearly, so the pairs too, so pairs' components too. Thus, we wrap the nodes of the input tree in the `Ur` wrapper, whose linear consumption allows for unrestricted use of its inner value, as detailed in Section 4.2.

This example shows how destinations can be used even in a non-linear setting in order to improve the expressiveness of the language. This more natural and less convoluted implementation of breadth-first traversal also presents great performance gains compared to the fancy functional implementation from [Gib+23], as detailed in Section 4.6.

Deserializing, lifetime, and garbage collection

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

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

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

However, we are still allocating two copies of the deserialized payload. This is wasteful, it would be much better to allocate directly in the region, but this isn't part of the original compact region API. Destinations are one way to accomplish this. In fact, as I'll explain in Section 4.5, my implementation of DPS for Haskell is backed by compact regions because they provide more freedom to do low-level memory operations without interfering with GC.

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

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

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

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

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

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

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

- even for error cases, we are forced to consume the destination that we receive as an argument (to stay linear), hence we write some sensible default data to it (see line 3);

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

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

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

Thanks to that new implementation which is barely longer (in terms of lines of code) than the naive one, the program runs almost twice as fast, mostly because garbage-collection time goes to almost zero. The detailed benchmark is available in Section 4.6.

4.4 API Design

Table 5 presents my pure API for functional DPS programming. This API is sufficient to implement all the examples of Section 4.3. This section explains its various parts in detail.

The `Incomplete` type

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

```
parseSListDPS :: ByteString → Int → Dest [SExpr] ⊸ Either Error Int
parseSListDPS bs i d = case bs !? i of
  Nothing → fill @'[] d ; Left (UnexpectedEOFSList i)
  Just x → if
    | x == ')' → fill @'[] d ; Right i
    | isSpace x → parseSListDPS bs (i + 1) d
    | otherwise →
       case fill @'(:) d of
        (dh, dt) → case parseSExprDPS bs i dh of
           Left err → fill @'[] dt ; Left err
           Right endPos → parseSListDPS bs (endPos + 1) dt
```

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

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

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

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

Listing 5: Destination API for Haskell

Hence we introduce a new data type `Incomplete a b` where `a` stands for the type of the structure being built, and `b` is the type of what needs to be linearly consumed before the structure can be read. The idea is that one can map over the `b` side, which will contain destinations or containers with destinations inside, until there is no destination left but just a non-linear value that can safely escape (e.g. `()`, `Int`, or something wrapped in `Ur`). When destinations from the `b` side are consumed, the structure on the `a` side is built little by little in a top-down fashion, as we showed in Figures 4.3 and 4.4. And when no destination remains on the `b` side, the value of type `a` no longer has holes, thus is ready to be released/read.

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

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

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

Ensuring write-once model for holes with linear types

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

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

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

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

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

Still, this is not enough; because `\x → x` is indeed a linear continuation, one could use `withR (\x → x)` to leak a `Resource`, and then use it in a non-linear fashion in the outside world. Hence we must forbid the resource from appearing anywhere in the return type of the continuation. To do that, we ask the return type to be wrapped in `Ur`: because the resource comes from the left of a linear arrow, and doesn't implement `Movable`, it cannot be wrapped in `Ur` without breaking linearity (see Section 4.2). On the other hand, a `Movable` value of type `()` or `Int` can be returned:

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

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

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

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

`fillLeaf :: ∀ a. a → Dest a ⊸ ()` uses a value of type `a` to fill the hole represented by the destination. The destination is consumed linearly, but the value to fill the hole isn't (as indicated by the first non-linear arrow). Memory-wise, the address of the object `a` is written into the memory cell pointed to by the destination (see Figure 4.7).

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

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

[2] This is why the `fromIncomplete'` and `fromIncomplete_'` variants aren't that useful in the actual memory-safe API (which differs slightly from the simplified examples of Section 4.3): here the built structure would be stuck in the scope function without its `Ur` escape pass.

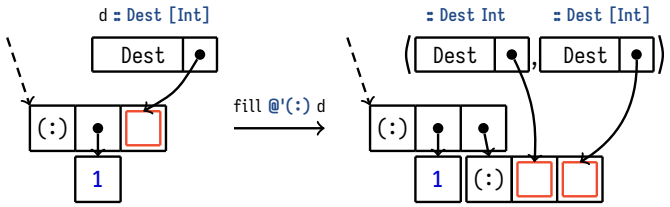


Figure 4.5: Memory behavior of `fill @'(:) :: Dest [a] ⊸ (Dest a, Dest [a])`

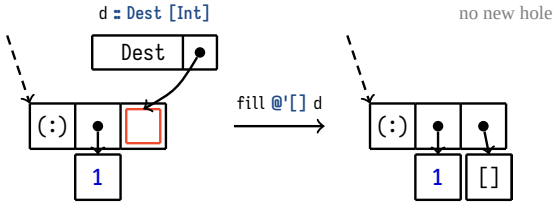


Figure 4.6: Memory behavior of `fill @'[] :: Dest [a] ⊸ ()`

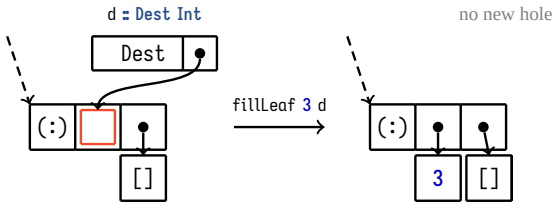


Figure 4.7: Memory behavior of `fillLeaf :: a → Dest [a] ⊸ ()`

The resulting structure `comp` is morally a `BigStruct` like `parent`, that inherited the hole from the child structure (`Dest Int`) and still has its other hole (`Dest OtherStruct`) to be filled. An example of memory behavior of `fillComp` in action can be seen in Figure 4.4.

`fill :: ∀ lCtor a. Dest a ⊸ DestsOf lCtor a` lets us build structures using layers of hollow constructors. It takes a constructor as a type parameter (`lCtor`) and allocates a hollow heap object that has the same header/tag as the specified constructor but unspecified fields. The address of the allocated hollow constructor is written in the destination that is passed to `fill`. As a result, one hole is now filled, but there is one new hole in the structure for each field left unspecified in the hollow constructor that is now part of the bigger structure. So `fill` returns one destination of matching type for each of the fields of the constructor. An example of the memory behavior of `fill @'(:) :: Dest [a] ⊸ (Dest a, Dest [a])` is given in Figure 4.5 and the one of `fill @'[] :: Dest [a] ⊸ ()` is given in Figure 4.6.

`DestsOf` is a type family (i.e. a function operating on types) whose role is to map a constructor to the type of destinations for its fields. For example, `DestsOf '[] [a] = ()` and `DestsOf '(:) [a] = (Dest a, Dest [a])`. More generally, there is a duality between the type of a constructor `Ctor :: (f1 ... fn) → a` and of the associated destination-filling functions `fill @'Ctor :: Dest a ⊸ (Dest f1 ... Dest fn)`. Destination-based data building can be seen as more general than the usual bottom-up constructor approach, as we can recover `Ctor` from the associated function `fill @'Ctor`, but not the reverse:

```
1 Ctor :: (f1 ... fn) → a
2 Ctor (x1 ... xn) = fromIncomplete_' (
3   alloc <&> \(d :: Dest a) → case fill @'Ctor d of
4     (dx1 ... dxn) → fillLeaf x1 dx1 ⨟ ... ⨟ fillLeaf xn dxn)
```

4.5 Implementing destinations in Haskell

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

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

Compact Regions

As I teased in Section 4.3, *compact regions* from [Yan+15] make it very convenient to implement DPS programming in Haskell. A compact region represents a memory area in the Haskell heap that is almost fully independent from the GC and the rest of the garbage-collected heap. For the GC, each compact region is seen as a single heap object with a single lifetime. The GC can efficiently check whether there is at least one pointer in the garbage-collected heap that points into the region, and while this is the case, the region is kept alive. When this condition is no longer matched, the whole region is discarded. The result is that the GC won't traverse any node from the region: it is treated as one opaque block (even though it is actually implemented as a chain of blocks of the same size, that doesn't change the principle). Also, compact regions are immobile in memory; the GC won't move them, so a destination can just be implemented as a raw pointer (type `Addr#` in Haskell): `data Dest r a = Dest Addr#`

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

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

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

The `withRegion` function is the new addition to the modified API presented in Listing 6 (the `Token` type and its associated functions `dup2` and `consume` are unchanged). `withRegion` is mostly a refinement over the `withToken` function from Listing 5. It receives a continuation in which `r` must be a free type variable. It then spawns both a new compact region and a fresh type `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 calls the continuation at type `r`. This is fairly standard practice since [LP94].

Representation of `Incomplete` objects

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

It is also natural for `alloc` to return an `Incomplete r a (Dest a)`: there is nothing more here than an empty memory cell (named *root receiver*) of type `a` which the associated destination of type `Dest a` points to, as presented in Figure 4.1. A bit like the identity function, whatever goes in the hole is exactly what will be retrieved in the `a` side.

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

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

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

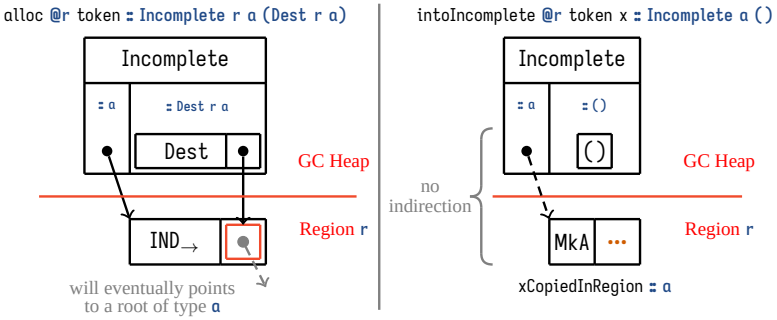


Figure 4.8: Memory behaviour of `alloc` and `intoIncomplete` in the region implementation

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

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

The desired outcome is to only allocate a root receiver in the region for actual incomplete structures, and skip that allocation for already complete structures that are turned into an `Incomplete` object, while preserving a same type for both use-cases. This is made possible by replacing the `Ur` wrapper inside the `Incomplete` by an indirection object (`stg_IND` label) for the actually-incomplete case. `Incomplete r a b` will be represented by a pair `(a, b)` allocated in the garbage-collected heap, with slight variations as illustrated in Figure 4.8:

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

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

Deriving `fill` for all constructors with `Generics`

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

In Section 4.5, we will see how to allocate a hollow heap object for a specified constructor (which is known at compile-time). The only other information we need to implement `fill` generically is the shape of the constructor, and more precisely the number and type of its fields. So we will leverage `GHC.Generics` to find the required information.

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

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

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

With a bit of type-level programming[3], we can extract the parts of that representation which are related to the constructor `lCtor` and use them inside the instance head of `Fill lCtor a` so the implementation of `fill` can depend on them. That's how we can give the proper types to the destinations returned by that function for a specified constructor. The `DestsOf lCtor a :: Type` type family also uses the generic representation of `a` to extract what it needs to know about `lCtor` and its fields.

Changes to GHC internals and RTS

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

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

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

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

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

- one *external primop* to allocate space inside a compact region for a hollow constructor. This primop has to be implemented inside the RTS for the aforementioned reasons;

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

- one *internal primop* (internal primops are macros which generates C– code) that will be resolved into a normal albeit static value representing the info table pointer of a given constructor. This value will be passed as an argument to the external primop.

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

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

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

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

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

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

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

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

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

4.6 Evaluating the performance of DPS programming

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

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

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

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

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

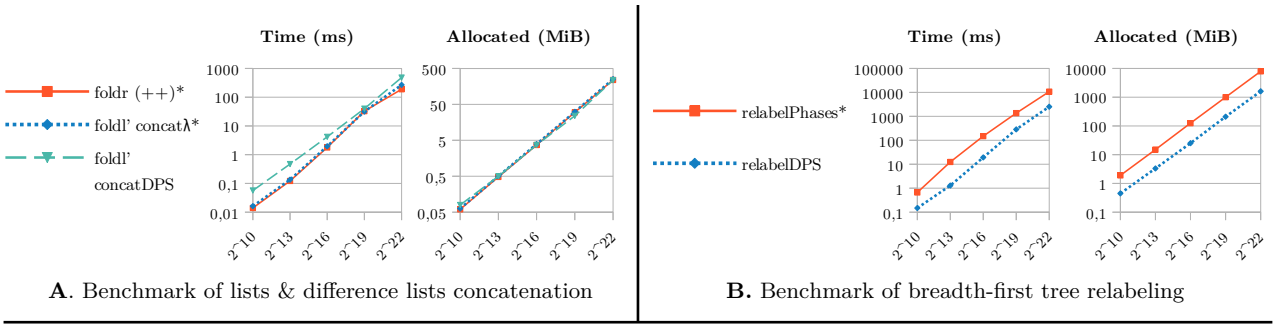
Listing 7: compactAddHollow# implementation in rts/Compact.cmm

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

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

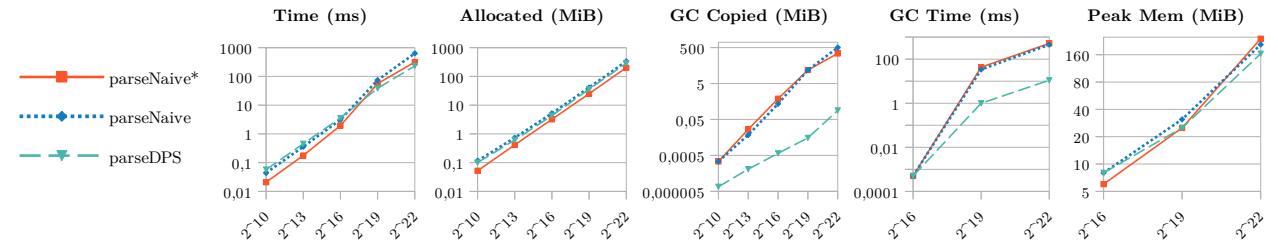
```
matchFamLCtorToSymbol :: [Type] → Maybe (CoAxiomRule, [Type], Type)
matchFamLCtorToSymbol [kind, ty]
  | TyConApp tyCon _ ← ty, Just dataCon ← isPromotedDataCon_maybe tyCon =
     let symbolLit = (mkStrLitTy . occNameFS . occName . getName $ dataCon)
      in Just (axLCtorToSymbolDef, [kind, ty],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

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

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



A. Benchmark of lists & difference lists concatenation

B. Benchmark of breadth-first tree relabeling



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

C. Benchmark of S-expressions parser

Figure 4.9: Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, `-N1 -O2`)

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

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

We see in part **A** of Figure 4.9 that the destination-backed difference lists have a comparable memory use as the two other linear implementations, while being quite slower (by a factor 2-4) on all datasets. We would expect better results though for a DPS implementation outside of compact regions because those cause extra copying.

Breadth-first relabeling We see in part **B** of Figure 4.9 that the destination-based tree traversal is almost one order of magnitude more efficient, both time-wise and memory-wise, compared to the implementation based on *Phases* applicatives presented in [Gib+23].

Parsing S-expressions In part **C** of Figure 4.9, we compare the naive implementation of the S-expression parser and the DPS one (see Section 4.3). For this particular program, where using compact regions might reduce the future GC load of the application, it is relevant to benchmark the naive version twice: once with `force` and once with `compact`.

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

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

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

4.7 Related work

The idea of functional data structures with write-once holes is not new. Minamide already proposed in [Min98] a variant of λ-calculus with support for *hole abstractions*, which can be represented in memory by an incomplete structure with one hole and can be composed efficiently with each other (as with `fillComp` in Figure 4.4). With such a framework, it is fully possible to implement destination-backed difference lists for example.

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

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

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

4.8 Conclusion and future work

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

There are two limitations that we would like to lift in the future. First, DPS programming could be useful outside of compact regions: destinations could probably be used to manipulate the garbage-collected heap (with proper read barriers in place), or other forms of secluded memory areas that aren't traveled by the GC (RDMA, network serialized buffers, etc.). Secondly, at the moment, the type of `fillLeaf` implies that we can't store

destinations (which are always linear) in a difference list implemented as in Section 4.3, whereas we can store them in a regular list or queue (like we do, for instance, in Section 4.3). This unwelcome restriction ensures memory safety but it's quite coarse grain. In the future we'll be trying to have a more fine-grained approach that would still ensure safety.

Chapter 5

Extending DPS support for linear data structures

TBD.

Chapter 6

Related work

TBD.

Chapter 7

Conclusion

TBD.

List of Figures

List of Listings

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[OLE19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. “Quantitative program reasoning with graded modal types.” In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). doi: 10.1145/3341714. url: https://doi.org/10.1145/3341714.

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Full reduction rules for λ_d

TODO: Fix overflowing figure

$$\frac{\mathtt{NotVal}\ t}{E\,[\,t'\ t\,] \longrightarrow (E \circ t'\ \square)\,[\,t\,]}\ \text{Focus-App1} \qquad \frac{}{(E \circ t'\ \square)\,[\,v\,] \longrightarrow E\,[\,t'\ v\,]}\ \text{Unfocus-App1}$$

$$\frac{\mathtt{NotVal}\ t'}{E\,[\,t'\ v\,] \longrightarrow (E \circ \square\ v)\,[\,t'\,]}\ \text{Focus-App2} \qquad \frac{}{(E \circ \square\ v)\,[\,v'\,] \longrightarrow E\,[\,v'\ v\,]}\ \text{Unfocus-App2}$$

$$\frac{}{E\,[\,(\lambda x_{\,m} \mapsto u)\ v\,] \longrightarrow E\,[\,u[x := v]\,]}\ \text{Red-App} \qquad \frac{\mathtt{NotVal}\ t}{E\,[\,t\ ;\ u\,] \longrightarrow (E \circ \square\ ;\ u)\,[\,t\,]}\ \text{Focus-PatU}$$

$$\frac{}{(E \circ \square\ ;\ u)\,[\,v\,] \longrightarrow E\,[\,v\ ;\ u\,]}\ \text{Unfocus-PatU} \qquad \frac{}{E\,[\,()\ ;\ u\,] \longrightarrow E\,[\,u\,]}\ \text{Red-PatU}$$

$$\frac{\mathtt{NotVal}\ t}{E\,[\,\mathtt{case}_m\ t\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow (E \circ \mathtt{case}_m\ \square\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\})\,[\,t\,]}\ \text{Focus-PatS}$$

$$\frac{}{(E \circ \mathtt{case}_m\ \square\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\})\,[\,v\,] \longrightarrow E\,[\,\mathtt{case}_m\ v\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,]}\ \text{Unfocus-PatS}$$

$$\frac{}{E\,[\,\mathtt{case}_m\ (\mathtt{Inl}\,v_1)\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_1[x_1 := v_1]\,]}\ \text{Red-PatL}$$

$$\frac{}{E\,[\,\mathtt{case}_m\ (\mathtt{Inr}\,v_2)\ \mathtt{of}\ \{\mathtt{Inl}\,x_1 \mapsto u_1\,,\ \mathtt{Inr}\,x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_2[x_2 := v_2]\,]}\ \text{Red-PatR}$$

$$\frac{\mathtt{NotVal}\ t}{E\,[\,\mathtt{case}_m\ t\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto u\,] \longrightarrow (E \circ \mathtt{case}_m\ \square\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto u)\,[\,t\,]}\ \text{Focus-PatP}$$

$$\frac{}{(E \circ \mathtt{case}_m\ \square\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto u)\,[\,v\,] \longrightarrow E\,[\,\mathtt{case}_m\ v\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto u\,]}\ \text{Unfocus-PatP}$$

$$\frac{}{E\,[\,\mathtt{case}_m\ (v_1\,,\,v_2)\ \mathtt{of}\ (x_1\,,\,x_2) \mapsto u\,] \longrightarrow E\,[\,u[x_1 := v_1][x_2 := v_2]\,]}\ \text{Red-PatP}$$

$$\frac{\mathtt{NotVal}\ t}{E\,[\,\mathtt{case}_m\ t\ \mathtt{of}\ \mathtt{Mod}_n\ x \mapsto u\,] \longrightarrow (E \circ \mathtt{case}_m\ \square\ \mathtt{of}\ \mathtt{Mod}_n\ x \mapsto u)\,[\,t\,]}\ \text{Focus-PatE}$$

$$\frac{}{(E \circ \mathtt{case}_m\ \square\ \mathtt{of}\ \mathtt{Mod}_n\ x \mapsto u)\,[\,v\,] \longrightarrow E\,[\,\mathtt{case}_m\ v\ \mathtt{of}\ \mathtt{Mod}_n\ x \mapsto u\,]}\ \text{Unfocus-PatE}$$

$$\frac{}{E\,[\,\mathtt{case}_m\ \mathtt{Mod}_n\ v'\ \mathtt{of}\ \mathtt{Mod}_n\ x \mapsto u\,] \longrightarrow E\,[\,u[x := v']\,]}\ \text{Red-PatE}$$

$$\frac{\mathtt{NotVal}\ t}{E\,[\,\mathtt{map}\ t\ \mathtt{with}\ x \mapsto t'\,] \longrightarrow (E \circ \mathtt{map}\ \square\ \mathtt{with}\ x \mapsto t')\,[\,t\,]}\ \text{Focus-Map}$$

$$\frac{}{(E \circ \mathtt{map}\ \square\ \mathtt{with}\ x \mapsto t')\,[\,v\,] \longrightarrow E\,[\,\mathtt{map}\ v\ \mathtt{with}\ x \mapsto t'\,]}\ \text{Unfocus-Map}$$

$$\frac{h''' = \mathtt{max}(H \cup \mathtt{hnames}(E)) + 1}{E\,[\,\mathtt{map}_H \langle v_2 \wedge v_1 \rangle\ \mathtt{with}\ x \mapsto t'\,] \longrightarrow (E \circ {}^{\mathrm{op}}_{H \pm h'''}\langle v_2[H \pm h'''] \wedge \square \rangle)\,[\,t'[x := v_1[H \pm h''']]\,]}\ \text{Open-Ampar}$$

$$\frac{}{(E \circ {}^{\mathrm{op}}_{H}\langle v_2 \wedge \square \rangle)\,[\,v_1\,] \longrightarrow E\,[\,{}_H\langle v_2 \wedge v_1 \rangle\,]}\ \text{Close-Ampar}$$

Figure A.1: Full reduction rules for λ_d (part 1)

$$\frac{\mathtt{NotVal}\ u}{E\left[\mathbf{to}_{\ltimes}\ u\right] \longrightarrow \left(E \circ \mathbf{to}_{\ltimes}\ \square\right)\left[u\right]}\ \text{Focus-ToA} \qquad \frac{}{\left(E \circ \mathbf{to}_{\ltimes}\ \square\right)\left[v_2\right] \longrightarrow E\left[\mathbf{to}_{\ltimes}\ v_2\right]}\ \text{Unfocus-ToA}$$

$$\frac{}{E\left[\mathbf{to}_{\ltimes}\ v_2\right] \longrightarrow E\left[\{\}\langle v_2 \wedge ()\rangle\right]}\ \text{Red-ToA} \qquad \frac{\mathtt{NotVal}\ t}{E\left[\mathbf{from}_{\ltimes}\ t\right] \longrightarrow \left(E \circ \mathbf{from}_{\ltimes}\ \square\right)\left[t\right]}\ \text{Focus-FromA}$$

$$\frac{}{\left(E \circ \mathbf{from}_{\ltimes}\ \square\right)\left[v\right] \longrightarrow E\left[\mathbf{from}_{\ltimes}\ v\right]}\ \text{Unfocus-FromA}$$

$$\frac{}{E\left[\mathbf{from}_{\ltimes}\ \{\}\langle v_2 \wedge \mathtt{Mod}_{1\infty}\ v_1\rangle\right] \longrightarrow E\left[\left(v_2\,,\ \mathtt{Mod}_{1\infty}\ v_1\right)\right]}\ \text{Red-FromA} \qquad \frac{}{E\left[\mathtt{alloc}\right] \longrightarrow E\left[\{1\}\langle \boxed{1} \wedge \to 1\rangle\right]}\ \text{Red-Alloc}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft ()\right] \longrightarrow \left(E \circ \square \triangleleft ()\right)\left[t\right]}\ \text{Focus-FillU} \qquad \frac{}{\left(E \circ \square \triangleleft ()\right)\left[v\right] \longrightarrow E\left[v \triangleleft ()\right]}\ \text{Unfocus-FillU}$$

$$\frac{}{E\left[\to h \triangleleft ()\right] \longrightarrow E\left(\!\left|h :=_{\{\}}\ ()\right|\!\right)\left[()\right]}\ \text{Red-FillU} \qquad \frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft \mathtt{Inl}\right] \longrightarrow \left(E \circ \square \triangleleft \mathtt{Inl}\right)\left[t\right]}\ \text{Focus-FillL}$$

$$\frac{}{\left(E \circ \square \triangleleft \mathtt{Inl}\right)\left[v\right] \longrightarrow E\left[v \triangleleft \mathtt{Inl}\right]}\ \text{Unfocus-FillL}$$

$$\frac{h' = \mathtt{max}(\mathtt{hnames}(E) \cup \{h\}) + 1}{E\left[\to h \triangleleft \mathtt{Inl}\right] \longrightarrow E\left(\!\left|h :=_{\{h'+1\}}\ \mathtt{Inl}\ \boxed{h'+1}\right|\!\right)\left[\to h'+1\right]}\ \text{Red-FillL}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft \mathtt{Inr}\right] \longrightarrow \left(E \circ \square \triangleleft \mathtt{Inr}\right)\left[t\right]}\ \text{Focus-FillR} \qquad \frac{}{\left(E \circ \square \triangleleft \mathtt{Inr}\right)\left[v\right] \longrightarrow E\left[v \triangleleft \mathtt{Inr}\right]}\ \text{Unfocus-FillR}$$

$$\frac{h' = \mathtt{max}(\mathtt{hnames}(E) \cup \{h\}) + 1}{E\left[\to h \triangleleft \mathtt{Inr}\right] \longrightarrow E\left(\!\left|h :=_{\{h'+1\}}\ \mathtt{Inr}\ \boxed{h'+1}\right|\!\right)\left[\to h'+1\right]}\ \text{Red-FillR}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft \mathtt{Mod}_{\mathtt{m}}\right] \longrightarrow \left(E \circ \square \triangleleft \mathtt{Mod}_{\mathtt{m}}\right)\left[t\right]}\ \text{Focus-FillE} \qquad \frac{}{\left(E \circ \square \triangleleft \mathtt{Mod}_{\mathtt{m}}\right)\left[v\right] \longrightarrow E\left[v \triangleleft \mathtt{Mod}_{\mathtt{m}}\right]}\ \text{Unfocus-FillE}$$

$$\frac{h' = \mathtt{max}(\mathtt{hnames}(E) \cup \{h\}) + 1}{E\left[\to h \triangleleft \mathtt{Mod}_{\mathtt{m}}\right] \longrightarrow E\left(\!\left|h :=_{\{h'+1\}}\ \mathtt{Mod}_{\mathtt{m}}\ \boxed{h'+1}\right|\!\right)\left[\to h'+1\right]}\ \text{Red-FillE}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft (,)\right] \longrightarrow \left(E \circ \square \triangleleft (,)\right)\left[t\right]}\ \text{Focus-FillP} \qquad \frac{}{\left(E \circ \square \triangleleft (,)\right)\left[v\right] \longrightarrow E\left[v \triangleleft (,)\right]}\ \text{Unfocus-FillP}$$

$$\frac{h' = \mathtt{max}(\mathtt{hnames}(E) \cup \{h\}) + 1}{E\left[\to h \triangleleft (,)\right] \longrightarrow E\left(\!\left|h :=_{\{h'+1,h'+2\}}\ \left(\boxed{h'+1}\,,\ \boxed{h'+2}\right)\right|\!\right)\left[\left(\to h'+1\,,\ \to h'+2\right)\right]}\ \text{Red-FillP}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft (\lambda x_{\mathtt{m}} \mapsto u)\right] \longrightarrow \left(E \circ \square \triangleleft (\lambda x_{\mathtt{m}} \mapsto u)\right)\left[t\right]}\ \text{Focus-FillF}$$

$$\frac{}{\left(E \circ \square \triangleleft (\lambda x_{\mathtt{m}} \mapsto u)\right)\left[v\right] \longrightarrow E\left[v \triangleleft (\lambda x_{\mathtt{m}} \mapsto u)\right]}\ \text{Unfocus-FillF}$$

$$\frac{}{E\left[\to h \triangleleft (\lambda x_{\mathtt{m}} \mapsto u)\right] \longrightarrow E\left(\!\left|h :=_{\{\}}\ \text{`}\lambda x_{\mathtt{m}} \mapsto u\right|\!\right)\left[()\right]}\ \text{Red-FillF}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \triangleleft_{\mathbf{o}} t'\right] \longrightarrow \left(E \circ \square \triangleleft_{\mathbf{o}} t'\right)\left[t\right]}\ \text{Focus-FillComp1} \qquad \frac{}{\left(E \circ \square \triangleleft_{\mathbf{o}} t'\right)\left[v\right] \longrightarrow E\left[v \triangleleft_{\mathbf{o}} t'\right]}\ \text{Unfocus-FillComp1}$$

$$\frac{\mathtt{NotVal}\ t'}{E\left[v \triangleleft_{\mathbf{o}} t'\right] \longrightarrow \left(E \circ v \triangleleft_{\mathbf{o}} \square\right)\left[t'\right]}\ \text{Focus-FillComp2} \qquad \frac{}{\left(E \circ v \triangleleft_{\mathbf{o}} \square\right)\left[v'\right] \longrightarrow E\left[v \triangleleft_{\mathbf{o}} v'\right]}\ \text{Unfocus-FillComp2}$$

$$\frac{h'' = \mathtt{max}(H \cup (\mathtt{hnames}(E) \cup \{h\})) + 1}{E\left[\to h \underset{H}{\triangleleft_{\mathbf{o}}} \left(v_2 \wedge v_1\right)\right] \longrightarrow E\left(\!\left|h :=_{(H \pm h'')}\ v_2[H \pm h'']\right|\!\right)\left[v_1[H \pm h'']\right]}\ \text{Red-FillComp}$$

$$\frac{\mathtt{NotVal}\ t}{E\left[t \blacktriangleleft t'\right] \longrightarrow \left(E \circ \square \blacktriangleleft t'\right)\left[t\right]}\ \text{Focus-FillLeaf1} \qquad \frac{}{\left(E \circ \square \blacktriangleleft t'\right)\left[v\right] \longrightarrow E\left[v \blacktriangleleft t'\right]}\ \text{Unfocus-FillLeaf1}$$

$$\frac{\mathtt{NotVal}\ t'}{\ldots}$$