Destination calculus

A linear λ−calculus for purely functional memory writes

ANONYMOUS AUTHOR(S)

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.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Formal language definitions**; **Functional languages**; **Data types and structures**.

Additional Key Words and Phrases: destination, functional programming, linear types, pure language

ACM Reference Format:
Anonymous Author(s). 2025. Destination calculus: A linear λ−calculus for purely functional memory writes. *Proc. ACM Program. Lang.* XX, XX, Article XXX (January 2025), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

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 $\mathsf{T} \to \mathsf{U}$ would, in destination-passing style, have type $\mathsf{T} \to \lfloor \mathsf{U} \rfloor \to 1$ instead. This style is common in systems programming, where destinations $\lfloor \mathsf{U} \rfloor$ are more commonly known as "out parameters". In C, $\lfloor \mathsf{U} \rfloor$ would typically be a pointer of type $\mathsf{U}*$.

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 U, it has to allocate the space for a U. But with destinations, the caller is responsible for finding space for a 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 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 [Shaikhha et al. 2017] 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 [Bagrel 2024].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2475-1421/2025/1-ARTXXX $15.00
https://doi.org/XXXXXXX.XXXXXXX

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 [Bagrel 2024] 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.

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 [Minamide 1998]. 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 [Bour et al. 2021];
- when destinations are first-class values, they can be passed and stored like ordinary values. This is the innovation of [Bagrel 2024] 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 5 and 6). λ_d is expressive enough so that previous calculi for destinations can be encoded in λ_d (see Section 9);
- a demonstration that λ_d is more expressive than previous calculi with destinations (Sections 3 and 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 [Bagrel 2024];
- an implementation strategy for λ_d which uses mutation without compromising the purity of λ_d (Section 8);
- formally-verified proofs, with the Coq proof assistant, of the main safety lemmas (Section 7).

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 $\mathsf{T} \oplus \mathsf{U}$ and $\mathsf{T} \otimes \mathsf{U}$ for sums and products, since λ_d is linearly typed, even though it isn't a focus in this section.

2.1 Building up a vocabulary

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

$$\mathbf{dId} : \mathsf{T} \to \lfloor \mathsf{T} \rfloor \to 1$$
$$\mathbf{dId}\ x\ d \triangleq d \blacktriangleleft x$$

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.

$$\textbf{fillWithInlCtor} : \lfloor T \oplus U \rfloor \to \lfloor T \rfloor$$
$$\textbf{fillWithInlCtor}\ d \triangleq d \triangleleft \mathsf{Inl}$$

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

Notice that with **fillWithInlCtor** we are constructing the structure from the outermost constructor inward: we've written a value of the form Inl □ into a hole, but we have yet to describe what goes in the new hole □. Such data constructors with uninitialized arguments are called *hollow constructors*. This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we make a value v and only then can we use Inl to make an 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 8). 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:

$$\textbf{fillWithPairCtor} : \lfloor T \otimes U \rfloor \to \lfloor T \rfloor \otimes \lfloor U \rfloor$$
$$\textbf{fillWithPairCtor}\ d \triangleq d \triangleleft (,)$$

After using **fillWithPairCtor**, the user must fill both the first field *and* the second field, using the destinations of type $\lfloor T \rfloor$ and $\lfloor U \rfloor$ 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 $\lfloor T \rfloor$ is a pointer to a yet-undefined part of a bigger structure. We say that such a structure has a hole of type 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 $T \otimes U$ that is being written to.

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

The general form $S \ltimes T$ is read "S ampar T". The name "ampar" stands for "asymmetric memory par"; we will explain why it is asymmetric in Section 5.2. For now, it's sufficient to observe that $S \ltimes \lfloor T \rfloor$ is akin to a "par" type $S \parr T^{\perp}$ in linear logic; you can think of $S \ltimes \lfloor T \rfloor$ as a (linear) function from T to S. That structures with holes could be seen a linear functions was first observed in [Minamide 1998], we elaborate on the value of having a par type rather than a function type in Section 4. A similar connective is called Incomplete in [Bagrel 2024].

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:

$$\begin{aligned}&\textbf{fillWithInlCtor}' \;:\; S \nrightarrow \lfloor T \oplus U \rfloor \to S \nrightarrow \lfloor T \rfloor\\ &\textbf{fillWithInlCtor}'\; x \triangleq \textbf{map}\; x \;\textbf{with}\; d \mapsto \textbf{fillWithInlCtor}\; d\end{aligned}$$

$$\begin{aligned}&\textbf{fillWithPairCtor}' \;:\; S \nrightarrow \lfloor T \otimes U \rfloor \to S \nrightarrow (\lfloor T \rfloor \otimes \lfloor U \rfloor)\\ &\textbf{fillWithPairCtor}'\; x \triangleq \textbf{map}\; x \;\textbf{with}\; d \mapsto \textbf{fillWithPairCtor}\; d\end{aligned}$$

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 $T \nrightarrow \lfloor T \rfloor$. alloc is a bit like the identity function: it is a hole (of type T) that needs a value of type T to be a complete value of type T. Memory-wise, it is an uninitialized block large enough to host a value of type T, and a destination pointing to it. Conversely, structures with holes are eliminated with[1] $\textbf{from}'_{\nrightarrow} : S \nrightarrow 1 \to S$: if all the destinations have been consumed and only unit remains on the right side, then 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.

$$\begin{aligned}&\mathsf{Inl} \;:\; T \to T \oplus U\\ &\mathsf{Inl}\; x \triangleq \textbf{from}'_{\nrightarrow}(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft x)\end{aligned}$$

$$\begin{aligned}&(,) \;:\; T \to U \to T \otimes U\\ &(x, y) \triangleq \textbf{from}'_{\nrightarrow}(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto \textbf{case}\; (d \triangleleft (,))\; \textbf{of}\; (d_1, d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y)\end{aligned}$$

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:

  $$\begin{aligned}&\textbf{forget} \;:\; T\\ &\textbf{forget} \triangleq \textbf{from}'_{\nrightarrow}(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto ())\end{aligned}$$

  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:

  $$\begin{aligned}&\textbf{ambiguous1} \;:\; \mathsf{Bool}\\ &\textbf{ambiguous1} \triangleq \textbf{from}'_{\nrightarrow}(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto d \blacktriangleleft \mathsf{true} \fatsemi d \blacktriangleleft \mathsf{false})\end{aligned}$$

  $$\begin{aligned}&\textbf{ambiguous2} \;:\; \mathsf{Bool}\\ &\textbf{ambiguous2} \triangleq \textbf{from}'_{\nrightarrow}(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto \textbf{let}\; x := (d \blacktriangleleft \mathsf{false})\; \textbf{in}\; d \blacktriangleleft \mathsf{true} \fatsemi x)\end{aligned}$$

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

[1] As the name suggest, there is a more general elimination $\textbf{from}_{\nrightarrow}$. It will be discussed in Section 5.

2.2 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 $X \mapsto 1 \oplus (T \otimes X)$. For convenience, we also define filling operators $\triangleleft[]$ and $\triangleleft(::)$:

$$\mathsf{List}\ T \overset{\text{rec}}{\triangleq} 1 \oplus (T \otimes (\mathsf{List}\ T))$$

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

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

Just like we did in Section 2.1 we can recover traditional constructors from filling operators:

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

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 [Hughes 1986]. 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 $x::\square$. Concatenation is filling the hole with another difference list, using operator $\triangleleft\circ$. The details are on the left of Figure 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*) [Hood and Melville 1981]. 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 (**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 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

```
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′ᵏ (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))
        }
    }
```

Fig. 1. Difference list and queue implementation in equirecursive λ_d

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 SCOPE ESCAPE OF DESTINATIONS

In Section 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 ◀ or ◃, 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 $\mathsf{T} \ast\ast$ 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 2.1, 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 **alloc′** operator[2]

```
alloc′ : (⌊T⌋ → 1) → T
alloc′ f ≜ from′ᵏ (map alloc with d ↦ f d)
```

[2]The actual semantics that we'll develop in Section 6 is more refined and can give a semantics to the more flexible **from′ᵏ** t and alloc operators directly, but for now, this simpler semantic will suffice for **alloc′**.

The semantics of **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:

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

For instance:

$$\begin{array}{rl} & \{\} \mid \textbf{alloc}'\ (\lambda d \mapsto d \triangleleft \mathsf{Inl} \triangleleft ()) \\ \longrightarrow & \{h := \square\} \mid \to h \triangleleft \mathsf{Inl} \triangleleft () \fatsemi \textbf{deref} \to h \\ \longrightarrow & \{h := \mathsf{Inl}\,()\} \mid \textbf{deref} \to h \\ \longrightarrow & \{\} \mid \mathsf{Inl}\,() \end{array}$$

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

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

Here $dd : \lfloor\lfloor \mathsf{Bool} \rfloor\rfloor$ and $d : \lfloor \mathsf{Bool} \rfloor$. 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:

$$\begin{array}{rl} & \{\} \mid \textbf{alloc}'\ (\lambda dd \mapsto \textbf{case}\ (\textbf{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d))\ \textbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\}) \\ \longrightarrow & \{hd := \square\} \mid \textbf{case}\ (\textbf{alloc}'\ (\lambda d \mapsto \to hd \blacktriangleleft d))\ \textbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\} \fatsemi \textbf{deref} \to hd \\ \longrightarrow & \{hd := \square, h := \square\} \mid \textbf{case}\ (\to hd \blacktriangleleft \to h \fatsemi \textbf{deref} \to h)\ \textbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\} \fatsemi \textbf{deref} \to hd \\ \longrightarrow & \{hd := \to h, h := \square\} \mid \textbf{case}\ (\textbf{deref} \to h)\ \textbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\} \fatsemi \textbf{deref} \to hd \end{array}$$

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

This example must be rejected by our type system. As demonstrated by [Bagrel 2024], it's possible to reject this example using purely a linear type system: they make it so that, in $d \blacktriangleleft \mathrm{t}$, 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 $\lfloor\lfloor \mathrm{T} \rfloor\rfloor$ practically useless: we can form destinations with type $\lfloor\lfloor \mathrm{T} \rfloor\rfloor$ 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 2.2.

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 2. So we want t in $d \blacktriangleleft \mathrm{t}$ 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 5.

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 [Bagrel 2024], 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 [Minamide 1998] 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 [Gibbons 1993; Gibbons et al. 2023; Okasaki 2000].

On the other hand, as demonstrated in [Bagrel 2024], 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 subtree*).

Figure 2 presents the λ_d implementation of the breadth-first tree traversal. There, Tree T is defined unsurprisingly as Tree T $\triangleq$ $1 \oplus (\mathrm{T} \otimes ((\text{Tree T}) \otimes (\text{Tree T})))$; we refer to the constructors of

```
go : (S ω∞⟶ T1 ⟶ (!ω∞S)⊗T2) ω∞⟶ S ω∞⟶ Queue (Tree T1⊗⌊Tree T2⌋) ⟶ 1
go f st q ≜rec 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
                            (E 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′⅄ (map alloc with dtree ↦ go f st (singleton (tree, dtree)))
relabelDPS : Tree 1 1∞⟶ Tree Nat
relabelDPS tree ≜ mapAccumBFS (λst ω∞↦ λun ↦ un ⨟ (E ω∞ (succ st), st)) 1 tree
```

Fig. 2. Breadth-first tree traversal in destination-passing style

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 2.2.

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 [Bagrel 2024]. The difference is that, because they can't store destinations in structures with holes (see the discussion in Section 3), their implementation can't use the efficient queue implementation from Section 2.2. 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 2 are the fuchsia annotations: these are *modes*. We'll describe modes in detail in Section 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 5.3 would reject the example. Arguments with no modes are otherwise linear and can capture destinations. We introduce the exponential modality $!_m T$ 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 $(!_m T) \longrightarrow U \simeq T \,{}_m\!\longrightarrow U$.

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 $\lfloor_n T\rfloor$ and ampar $S \ltimes T$ types which we've introduced in Sections 2 to 4. Just as important are *modes* and the associated exponential modality $!_m$. The grammar of λ_d is presented in Figure 3. Some of the constructions that we've been using in Sections 2 to 4 are syntactic sugar for more fundamental forms, we give their definitions in Figure 4.

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

Fig. 3. Grammar of λ_d

$$
\begin{array}{l}
\mathsf{t} \blacktriangleleft \mathsf{t}' \triangleq \mathsf{t} \triangleleft\!\circ (\mathbf{to}_{\ltimes}\ \mathsf{t}') \\
() \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto d \triangleleft ()) \\
\mathsf{Inl}\, \mathsf{t} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad d \triangleleft \mathsf{Inl} \blacktriangleleft \mathsf{t} \\
\qquad) \\
\mathsf{Inr}\, \mathsf{t} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad d \triangleleft \mathsf{Inr} \blacktriangleleft \mathsf{t} \\
\qquad) \\
\mathsf{E}_{\mathsf{m}}\, \mathsf{t} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad d \triangleleft \mathsf{E}_{\mathsf{m}} \blacktriangleleft \mathsf{t} \\
\qquad)
\end{array}
\qquad
\begin{array}{l}
\mathbf{from}'_{\ltimes}\,\mathsf{t} \triangleq \\
\quad \mathbf{case}\ (\mathbf{from}_{\ltimes}\ (\mathbf{map}\ \mathsf{t}\ \mathbf{with}\ un \mapsto un \fatsemi \mathsf{E}_{1\infty}\ ()))\ \mathbf{of} \\
\qquad (st, ex) \mapsto \mathbf{case}\ ex\ \mathbf{of} \\
\qquad\quad \mathsf{E}_{1\infty}\ un \mapsto un \fatsemi st \\
\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u} \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad d \triangleleft (\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u}) \\
\qquad) \\
(\mathsf{t}_1, \mathsf{t}_2) \triangleq \mathbf{from}'_{\ltimes}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad \mathbf{case}\ (d \triangleleft (,))\ \mathbf{of} \\
\qquad\quad (d_1, d_2) \mapsto d_1 \blacktriangleleft \mathsf{t}_1 \fatsemi d_2 \blacktriangleleft \mathsf{t}_2 \\
\qquad)
\end{array}
$$

Fig. 4. Syntactic sugar for terms

Following a common trend (e.g. [Abel and Bernardy 2020; Atkey 2018; Bernardy et al. 2018; Orchard et al. 2019]), our modes form a semiring[3]. In λ_d the mode semiring is the product of a *multiplicity* semiring for linearity, as in [Bernardy et al. 2018], and of an *age* semiring (see Section 5.1) to prevent the scoping issues discussed in Section 3.

We usually omit mode annotations when the mode is the unit element 1ν. In particular, a function arrow without annotation, or with multiplicity annotation 1, is linear; it is equivalent to the linear arrow $\multimap$ from [Girard 1995].

5.1 The age semiring

In order to prevent destinations from escaping their scope, as discussed in Section 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 5.3 that scopes are introduced by $\mathbf{map}\ \mathsf{t}\ \mathbf{with}\ x \mapsto \mathsf{t}'$.

[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 [Abel and Bernardy 2020]. So maybe a more accurate terminology would be commutative ringoids with units. We'll stick to "semiring" from now on.

$+$	$\uparrow^n$	∞
$\uparrow^m$	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	∞	∞

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

Fig. 5. Operation tables for age and multiplicity semirings

If we have a term **map** t_1 **with** $x_1 \mapsto$ **map** t_2 **with** $x_2 \mapsto$ **map** t_3 **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 **map** t **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.

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 5.

5.2 Design motivation behind the ampar and destination types

Minamide's work [Minamide 1998] 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 2.2.

In [Minamide 1998], 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 $(T, S)\mathsf{hfun}$ is thus a weak form of linear lambda abstraction $T \multimap S$, 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 $T \multimap S$ into $S \parr T^\perp$. Doing so for the type $(T, S)\mathsf{hfun}$ gives $S \parr \lfloor T \rfloor$, 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 $\lfloor T \rfloor$ — 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 $\parr$ to ampar $\between$. In CLL, thanks to the cut rule, any of the sides S or T of a par $S \parr T$ 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 T and $\lfloor T \rfloor$, and interaction between T and $T \to \bullet$. The structure containing holes, S, can safely interact with $\lfloor S \rfloor$ (merge it into another structure with holes), but not with $T \to \bullet$, as it would let us read an incomplete structure!

On the other hand, a complete value of type $T = (\ldots \lfloor T' \rfloor \ldots)$ containing destinations (but no holes) can safely interact with a function $T \to 1$: in particular, the function can pattern-match on the

value of type T to access destination $\lfloor \mathsf{T}' \rfloor$. However, it might not be safe to fill the $\mathsf{T} = (\ldots \lfloor \mathsf{T}' \rfloor \ldots)$ into a $\lfloor \mathsf{T} \rfloor$ as that might allow scope escape of the destination $\lfloor \mathsf{T}' \rfloor$ as we've just seen in Section 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 ($\mathsf{S} \ltimes \mathsf{T}$) for *asymmetrical memory par*:

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

5.3 Typing rules

The typing rules for λ_d are highly inspired from[Abel and Bernardy 2020] and Linear Haskell [Bernardy et al. 2018], and are detailed in Figure 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 $\mathsf{n}' \cdot (x :_{\mathsf{m}} \mathsf{T}) \triangleq x :_{\mathsf{n}' \cdot \mathsf{m}} \mathsf{T}$.
- We define context addition as a partial operation where $(x :_{\mathsf{m}} \mathsf{T}) + \Gamma = x :_{\mathsf{m}} \mathsf{T},\ \Gamma$ if $x \notin \Gamma$ and $(x :_{\mathsf{m}} \mathsf{T}) + (x :_{\mathsf{m}'} \mathsf{T}) = x :_{\mathsf{m}+\mathsf{m}'} \mathsf{T}$.

Figure 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 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 `DisposableOnly` Γ 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.

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

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

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 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 $!_{\omega \mathsf{a}}(\mathsf{T}_1 \otimes \mathsf{T}_2) \simeq (!_{\omega \mathsf{a}} \mathsf{T}_1) \otimes (!_{\omega \mathsf{a}} \mathsf{T}_2)$, which is not part of intuitionistic linear logic, but valid in Linear Haskell [Bernardy et al. 2018].

Rules for scoping. As destinations always exist in the context of a structure with holes, and must stay in that context (see Section 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** $x \mapsto \mathsf{t}'$ 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** $x \mapsto \mathsf{t}'$ 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 7 illustrates scopes introduced by **map**, and how the typing rules for **map** and $\blacktriangleleft$ interact.

Anticipating Section 6.1, 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,

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

$$\textsc{Ty-term-Var}\ \frac{\texttt{DisposableOnly}\ \Gamma \qquad 1_\nu <: m}{\Gamma,\ x :_m T \vdash x : T} \qquad \textsc{Ty-term-App}\ \frac{\Gamma_1 \vdash t : T \qquad \Gamma_2 \vdash t' : T \xrightarrow{m} U}{m\cdot\Gamma_1 + \Gamma_2 \vdash t'\ t : U} \qquad \textsc{Ty-term-PatU}\ \frac{\Gamma_1 \vdash t : 1 \qquad \Gamma_2 \vdash u : U}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : U}$$

$$\textsc{Ty-term-PatS}\ \frac{\Gamma_1 \vdash t : T_1 \oplus T_2 \qquad \Gamma_2,\ x_1 :_m T_1 \vdash u_1 : U \qquad \Gamma_2,\ x_2 :_m T_2 \vdash u_2 : U}{m\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\} : U}$$

$$\textsc{Ty-term-PatP}\ \frac{\Gamma_1 \vdash t : T_1 \otimes T_2 \qquad \Gamma_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{m\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ (x_1, x_2) \mapsto u : U}$$

$$\textsc{Ty-term-PatE}\ \frac{\Gamma_1 \vdash t : !_n T \qquad \Gamma_2,\ x :_{m\cdot n} T \vdash u : U}{m\cdot\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ \mathsf{E}_n\ x \mapsto u : U} \qquad \textsc{Ty-term-Map}\ \frac{\Gamma_1 \vdash t : U \ltimes T \qquad 1\uparrow\cdot\Gamma_2,\ x :_{1_\nu} T \vdash t' : T'}{\Gamma_1 + \Gamma_2 \vdash \mathbf{map}\ t\ \mathbf{with}\ x \mapsto t' : U \ltimes T'} \qquad \textsc{Ty-term-ToA}\ \frac{\Gamma \vdash u : U}{\Gamma \vdash \mathbf{to}_{\ltimes}\ u : U \ltimes 1}$$

$$\textsc{Ty-term-FromA}\ \frac{\Gamma \vdash t : U \ltimes (!_{1_\infty} T)}{\Gamma \vdash \mathbf{from}_{\ltimes}\ t : U \otimes (!_{1_\infty} T)} \qquad \textsc{Ty-term-Alloc}\ \frac{\texttt{DisposableOnly}\ \Gamma}{\Gamma \vdash \mathsf{alloc} : T \ltimes \lfloor T \rfloor} \qquad \textsc{Ty-term-FillU}\ \frac{\Gamma \vdash t : \lfloor_n 1 \rfloor}{\Gamma \vdash t \triangleleft () : 1} \qquad \textsc{Ty-term-FillL}\ \frac{\Gamma \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inl} : \lfloor_n T_1 \rfloor}$$

$$\textsc{Ty-term-FillR}\ \frac{\Gamma \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inr} : \lfloor_n T_2 \rfloor} \qquad \textsc{Ty-term-FillP}\ \frac{\Gamma \vdash t : \lfloor_n T_1 \otimes T_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor} \qquad \textsc{Ty-term-FillE}\ \frac{\Gamma \vdash t : \lfloor_n !_{n'} T \rfloor}{\Gamma \vdash t \triangleleft \mathsf{E}_{n'} : \lfloor_{n'\cdot n} T \rfloor}$$

$$\textsc{Ty-term-FillF}\ \frac{\Gamma_1 \vdash t : \lfloor_n T \xrightarrow{m} U \rfloor \qquad \Gamma_2,\ x :_m T \vdash u : U}{\Gamma_1 + (1\uparrow\cdot n)\cdot\Gamma_2 \vdash t \triangleleft (\lambda x_m \mapsto u) : 1} \qquad \textsc{Ty-term-FillComp}\ \frac{\Gamma_1 \vdash t : \lfloor_n U \rfloor \qquad \Gamma_2 \vdash t' : U \ltimes T}{\Gamma_1 + (1\uparrow\cdot n)\cdot\Gamma_2 \vdash t \triangleleft\!\circ\, t' : T}$$

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

$$\textsc{Ty-sterm-FromA'}\ \frac{\Gamma \vdash t : T \ltimes 1}{\Gamma \vdash \mathbf{from}'_{\ltimes}\ t : T} \qquad \textsc{Ty-sterm-FillLeaf}\ \frac{\Gamma_1 \vdash t : \lfloor_n T \rfloor \qquad \Gamma_2 \vdash t' : T}{\Gamma_1 + (1\uparrow\cdot n)\cdot\Gamma_2 \vdash t \blacktriangleleft t' : 1} \qquad \textsc{Ty-sterm-Unit}\ \frac{\texttt{DisposableOnly}\ \Gamma}{\Gamma \vdash () : 1}$$

$$\textsc{Ty-sterm-Fun}\ \frac{\Gamma_2,\ x :_m T \vdash u : U}{\Gamma_2 \vdash \lambda x_m \mapsto u : T \xrightarrow{m} U} \qquad \textsc{Ty-sterm-Left}\ \frac{\Gamma_2 \vdash t : T_1}{\Gamma_2 \vdash \mathsf{Inl}\ t : T_1 \oplus T_2} \qquad \textsc{Ty-sterm-Right}\ \frac{\Gamma_2 \vdash t : T_2}{\Gamma_2 \vdash \mathsf{Inr}\ t : T_1 \oplus T_2} \qquad \textsc{Ty-sterm-Exp}\ \frac{\Gamma_2 \vdash t : T}{m\cdot\Gamma_2 \vdash \mathsf{E}_m\ t : !_m T}$$

$$\textsc{Ty-sterm-Prod}\ \frac{\Gamma_{21} \vdash t_1 : T_1 \qquad \Gamma_{22} \vdash t_2 : T_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1, t_2) : T_1 \otimes T_2}$$

Fig. 6. Typing rules for terms and syntactic sugar

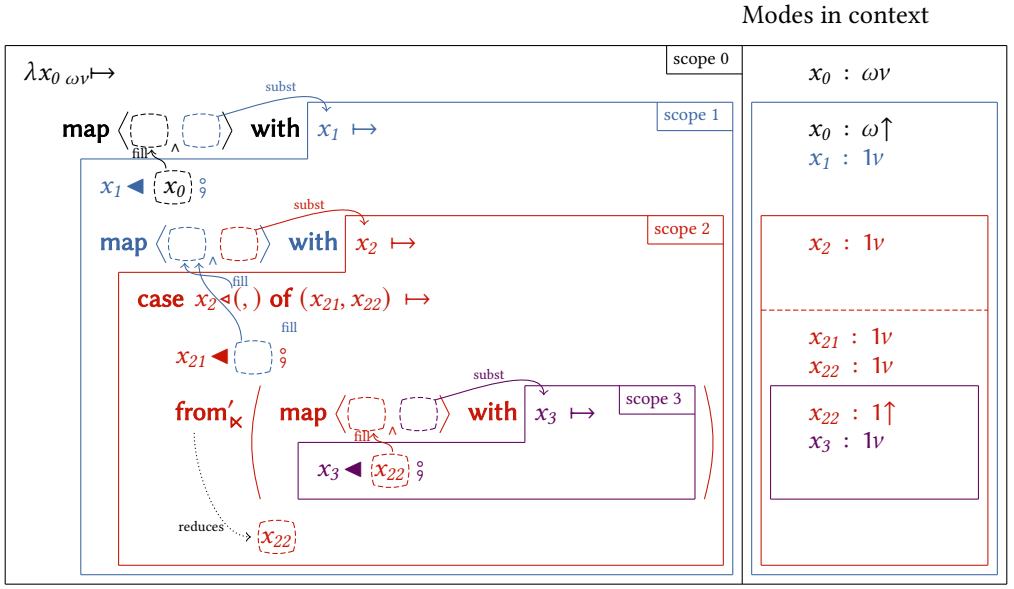

Fig. 7. Scope rules for **map** in λ_d

but the structure with holes remains in the outer scope. As a result, when filling a destination with rule Ty-sterm-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 $\blacktriangleleft$, and similarly, $\triangleleft\!\circ$ and $\triangleleft(\lambda x\ {}_{\mathsf{m}}\!\mapsto u)$, 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 $\mathbf{from}'_{\ltimes}$ (rule Ty-sterm-FromA'), the left of an ampar is extracted to the current scope (as seen at the bottom of Figure 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. $\mathbf{from}'_{\ltimes}$ is implemented in terms of $\mathbf{from}_{\ltimes}$ in Figure 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 $\mathbf{from}_{\ltimes}$ 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 $\mathbf{to}_{\ltimes}$ 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.

$$
\begin{array}{rcll}
\mathsf{t}, \mathsf{u} & ::= & \ldots \mid \mathsf{v} & \\
\\
\mathsf{v} & ::= & \boxed{h} & \textit{(hole)} \\
& \mid & \rightarrow h & \textit{(destination)} \\
& \mid & {}_{H}\langle \mathsf{v}_2 \,{}_{\wedge}\, \mathsf{v}_1 \rangle & \textit{(ampar value form)} \\
& \mid & () \mid {}^{\mathsf{v}}\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u} \mid \mathsf{Inl}\,\mathsf{v} \mid \mathsf{Inr}\,\mathsf{v} \mid \mathsf{E}_{\mathsf{m}}\,\mathsf{v} \mid (\mathsf{v}_1, \mathsf{v}_2) &
\end{array}
$$

$$
\begin{array}{rcllllllll}
\Delta & ::= & \cdot \mid \rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor & & & \mid \Delta_1, \Delta_2 & \mid \Delta_1 + \Delta_2 & \mid \mathsf{m}\cdot\Delta \\
\Gamma & ::= & \cdot \mid \rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor & \mid x :_{\mathsf{m}} \mathsf{T} & & \mid \Gamma_1, \Gamma_2 & \mid \Gamma_1 + \Gamma_2 & \mid \mathsf{m}\cdot\Gamma \\
\Theta & ::= & \cdot \mid \rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor & \mid \boxed{h} :_{\mathsf{n}} \mathsf{T} & \mid \rightarrow^{-1}\Delta & \mid \Theta_1, \Theta_2 & \mid \Theta_1 + \Theta_2 & \mid \mathsf{m}\cdot\Theta
\end{array}
$$

Fig. 8. Extended terms and runtime values

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

6 OPERATIONAL SEMANTICS

Before we define the operational semantics of λ_d we need to introduce a few more concepts. We'll need commands $\mathsf{C}\lceil \mathsf{t} \rceil$, they're described in Section 6.2; and we'll need values, described in 3. Indeed, the terms of λ_d lack any way to represent destinations or holes, or really any kind of value (for instance $\mathsf{Inl}\,()$ has been, so far, just syntactic sugar for a term $\mathbf{from}'_{\ltimes}\ (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ \ldots)$). 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.

6.1 Runtime values and extended terms

The syntax of runtime values is given in Figure 8. It features constructors for all of our basic types, as well as functions (note that in ${}^{\mathsf{v}}\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u}$, u is a term, not a value). The more interesting values are holes $\boxed{h}$, destinations $\rightarrow h$, and ampars ${}_{H}\langle \mathsf{v}_2 \,{}_{\wedge}\, \mathsf{v}_1 \rangle$, 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 2.1, and we must ensure that throughout the reduction, a destination always points to a hole, and a hole is always the target of exactly one destination. Thus, the new idea of our type system is to feature *hole bindings* $\boxed{h} :_{\mathsf{n}} \mathsf{T}$ and *destination bindings* $\rightarrow h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ in 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:

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

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

$$\textsc{Ty-term-Val}\ \frac{\texttt{DisposableOnly}\ \Gamma \qquad \Delta \overset{\nu}{\vdash} v : T}{\Gamma,\ \Delta \vdash v : T}$$

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

$$\textsc{Ty-val-Hole}\ \frac{}{\boxed{h} :_{1\nu} T \overset{\nu}{\vdash} \boxed{h} : T} \qquad \textsc{Ty-val-Dest}\ \frac{}{\rightarrow h :_{1\nu} \lfloor_n T \rfloor \overset{\nu}{\vdash} \rightarrow h : \lfloor_n T \rfloor} \qquad \textsc{Ty-val-Unit}\ \frac{}{\bullet \overset{\nu}{\vdash} () : 1} \qquad \textsc{Ty-val-Fun}\ \frac{\Delta,\ x :_m T \vdash u : U}{\Delta \overset{\nu}{\vdash} \lambda x \,{}_m\!\mapsto u : T \,{}_m\!\rightarrow U}$$

$$\textsc{Ty-val-Left}\ \frac{\Theta \overset{\nu}{\vdash} v_1 : T_1}{\Theta \overset{\nu}{\vdash} \mathsf{Inl}\, v_1 : T_1 \oplus T_2} \qquad \textsc{Ty-val-Right}\ \frac{\Theta \overset{\nu}{\vdash} v_2 : T_2}{\Theta \overset{\nu}{\vdash} \mathsf{Inr}\, v_2 : T_1 \oplus T_2} \qquad \textsc{Ty-val-Prod}\ \frac{\Theta_1 \overset{\nu}{\vdash} v_1 : T_1 \qquad \Theta_2 \overset{\nu}{\vdash} v_2 : T_2}{\Theta_1 + \Theta_2 \overset{\nu}{\vdash} (v_1, v_2) : T_1 \otimes T_2} \qquad \textsc{Ty-val-Exp}\ \frac{\Theta \overset{\nu}{\vdash} v' : T}{n \cdot \Theta \overset{\nu}{\vdash} \mathsf{E}_n\, v' : !_n T}$$

$$\textsc{Ty-val-Ampar}\ \frac{\texttt{LinOnly}\ \Delta_3 \qquad \texttt{FinAgeOnly}\ \Delta_3 \qquad 1\!\uparrow\!\cdot\Delta_1,\ \Delta_3 \overset{\nu}{\vdash} v_1 : T \qquad \Delta_2,\ (\rightarrow^{-1}\Delta_3) \overset{\nu}{\vdash} v_2 : U}{\Delta_1,\ \Delta_2 \overset{\nu}{\vdash} \mathsf{hnames}(\Delta_3)\langle v_2 \wedge v_1 \rangle : U \ltimes T}$$

Fig. 9. Typing rules for extended terms and runtime values

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

The last definition ranges over $\mathsf{name} ::= x \mid \boxed{h} \mid \rightarrow h$ and $\Omega ::= \Gamma \mid \Theta \mid \Delta$. Context addition is still very partial; for instance, $(\boxed{h} :_n T) + (\rightarrow h :_m \lfloor_{n'} T \rfloor)$ 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 9) maintains this invariant by simply not allowing any hole bindings in the context when typing terms (see Figure 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 ${}_H\langle v_2 \wedge v_1 \rangle$, in rule Ty-val-Ampar.

Specifically, holes come from the operator $\rightarrow^{-1}$, which represents the matching hole bindings for a set of destination bindings. It is a partial operation on destination-binding contexts defined pointwise as: $\rightarrow^{-1}(\rightarrow h :_{1\nu} \lfloor_n T \rfloor) = \boxed{h} :_n T$. Note that $\rightarrow^{-1}$ 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.

The properties $\texttt{LinOnly}\ \Delta_3$ and $\texttt{FinAgeOnly}\ \Delta_3$ are true given that $\rightarrow^{-1}\Delta_3$ 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.

$$
\begin{array}{rcl}
\mathsf{c} & ::= & \mathsf{t}'\ [] \ \mid\ [] \ \mathsf{v} \ \mid\ [] \fatsemi \mathsf{u} \\
& \mid & \mathbf{case}_{\mathsf{m}}\ [] \ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto \mathsf{u}_1\,,\ \mathsf{Inr}\, x_2 \mapsto \mathsf{u}_2\} \ \mid\ \mathbf{case}_{\mathsf{m}}\ [] \ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathsf{u} \ \mid\ \mathbf{case}_{\mathsf{m}}\ [] \ \mathbf{of}\ \mathsf{E}_{\mathsf{n}}\, x \mapsto \mathsf{u} \\
& \mid & \mathbf{map}\ [] \ \mathbf{with}\ x \mapsto \mathsf{t}' \ \mid\ \mathbf{to}_{\mathsf{K}}\ [] \ \mid\ \mathbf{from}_{\mathsf{K}}\ [] \\
& \mid & [] \triangleleft () \ \mid\ [] \triangleleft \mathsf{Inl} \ \mid\ [] \triangleleft \mathsf{Inr} \ \mid\ [] \triangleleft (,) \ \mid\ [] \triangleleft \mathsf{E}_{\mathsf{m}} \ \mid\ [] \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u}) \ \mid\ [] \triangleleft\!\!\circ\, \mathsf{t}' \ \mid\ \mathsf{v} \triangleleft\!\!\circ [] \\
& \mid & \overset{\mathsf{op}}{H}\langle \mathsf{v}_2 \wedge [] \rangle \qquad \textit{(open ampar focusing component)}
\end{array}
$$

$$
\mathsf{C} \ ::= \ [] \ \mid\ \mathsf{C} \circ \mathsf{c} \ \mid\ \mathsf{C}(\!|h :=_H \mathsf{v}|\!)
$$

Fig. 10. Grammar for evaluation contexts

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 $\lambda x \mapsto \mathbf{from}'_{\mathsf{K}}\,(\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto d \triangleleft (\lambda y \mapsto x\ \mathbf{concat}\ y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\lambda y \mapsto x\ \mathbf{concat}\ y)$ is part of term syntax, it's allowed to have free variable x.

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. $\mathsf{E}_{\omega\nu}\ \boxed{h}$. 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.

6.2 Evaluation contexts and commands

The semantics of λ_d is given using small-step reductions on a pair $\mathsf{C}[\mathsf{t}]$ of an evaluation context C, and an (extended) term t under focus. We call such a pair $\mathsf{C}[\mathsf{t}]$ a *command*, borrowing the terminology from [Curien and Herbelin 2000]. 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 $\mathsf{C}[\mathsf{t}]$ 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.

The grammar of evaluation contexts is given in Figure 10. An evaluation context C is the composition of an arbitrary number of focusing components $c_1, c_2 \ldots$. 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 $\overset{\text{op}}{H}\langle v_2 \wedge [] \rangle$. 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 $C\,(\!|h :=_H v|\!)$ that is used to update structures under construction that are attached to open ampar focusing components in the stack. Such a substitution is triggered when a destination $\to h$ is filled in the term under focus, 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:

$$\begin{array}{rcll} \left(C \circ \overset{\text{op}}{\{h\} \sqcup H}\langle v_2 \wedge [] \rangle\right)(\!|h :=_{H'} v'|\!) & = & C \circ \overset{\text{op}}{H \sqcup H'}\langle v_2\,(\!|h :=_{H'} v'|\!) \wedge [] \rangle & \\ (C \circ c)(\!|h :=_{H'} v'|\!) & = & C\,(\!|h :=_{H'} v'|\!) \circ c & \textit{otherwise} \end{array}$$

Evaluation contexts C 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 $C[t]$.

In other words, while $\Gamma \vdash t : T$ *requires* the binding in context Γ, $\Delta \dashv C : T \rightarrowtail U_0$ *provides* the bindings in Δ. The typing rules for evaluation contexts C and commands $C[t]$ are presented in Figure 11.

An evaluation context has a context type $T \rightarrowtail U_0$. The meaning of $C : T \rightarrowtail U_0$ is that given $t : T$, $C[t]$ returns a value of type U_0. Composing an evaluation context $C : T \rightarrowtail U_0$ with a new focusing component never affects the type U_0 of the future command; only the type 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:

$$\textsc{Ty-term-PatP}\ \frac{\Gamma_1 \vdash t : T_1 \otimes T_2 \qquad \Gamma_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{m \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_m\ t\ \mathbf{of}\ (x_1, x_2) \mapsto u : U} \qquad \textsc{Ty-ectxs-PatP}\ \frac{m \cdot \Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{\Delta_1 \dashv C \circ \mathbf{case}_m\ []\ \mathbf{of}\ (x_1, x_2) \mapsto u : (T_1 \otimes T_2) \rightarrowtail U_0}$$

- the typing context $m \cdot \Delta_1,\ \Delta_2$ in the premise for C corresponds to $m \cdot \Gamma_1 + \Gamma_2$ in the conclusion of Ty-term-PatP;
- the typing context $\Delta_2,\ x_1 :_m T_1,\ x_2 :_m T_2$ in the premise for term u corresponds to the typing context $\Gamma_2,\ x_1 :_m T_1,\ x_2 :_m T_2$ for the same term in Ty-term-PatP;
- the typing context Δ_1 in the conclusion for $C \circ \mathbf{case}_m\ []\ \mathbf{of}\ (x_1, x_2) \mapsto u$ corresponds to the typing context Γ_1 in the premise for t in Ty-term-PatP (the term t is located where the focus $[]$ is in Ty-ectxs-OpenAmpar).

$\boxed{\Delta \dashv C : T \rightarrowtail U_0}$ *(Typing judgment for evaluation contexts)*

$$\textsc{Ty-ectxs-Id}\ \frac{}{\cdot \dashv [] : U_0 \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-App1}\ \frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \quad \Delta_2 \vdash t' : T \to_m U}{\Delta_1 \dashv C \circ t'\ [] : T \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-App2}\ \frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \quad \Delta_1 \vdash v : T}{\Delta_2 \dashv C \circ []\ v : (T \to_m U) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-PatU}\ \frac{\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \quad \Delta_2 \vdash u : U}{\Delta_1 \dashv C \circ [] \fatsemi u : 1 \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-PatS}\ \frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \quad \Delta_2,\ x_1 :_m T_1 \vdash u_1 : U \quad \Delta_2,\ x_2 :_m T_2 \vdash u_2 : U}{\Delta_1 \dashv C \circ \mathbf{case}_m\ []\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : (T_1 \oplus T_2) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-PatP}\ \frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \quad \Delta_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{\Delta_1 \dashv C \circ \mathbf{case}_m\ []\ \mathbf{of}\ (x_1, x_2) \mapsto u : (T_1 \otimes T_2) \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-PatE}\ \frac{m{\cdot}\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \quad \Delta_2,\ x :_{m\cdot m'} T \vdash u : U}{\Delta_1 \dashv C \circ \mathbf{case}_m\ []\ \mathbf{of}\ \mathsf{E}_{m'}\ x \mapsto u : !_{m'} T \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-Map}\ \frac{\Delta_1,\ \Delta_2 \dashv C : U \ltimes T' \rightarrowtail U_0 \quad 1\uparrow{\cdot}\Delta_2,\ x :_{1_\nu} T \vdash t' : T'}{\Delta_1 \dashv C \circ \mathbf{map}\ []\ \mathbf{with}\ x \mapsto t' : (U \ltimes T) \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-ToA}\ \frac{\Delta \dashv C : (U \ltimes 1) \rightarrowtail U_0}{\Delta \dashv C \circ \mathbf{to}_{\ltimes}\ [] : U \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FromA}\ \frac{\Delta \dashv C : (U \otimes (!_{1_\infty} T)) \rightarrowtail U_0}{\Delta \dashv C \circ \mathbf{from}_{\ltimes}\ [] : (U \ltimes (!_{1_\infty} T)) \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-FillU}\ \frac{\Delta \dashv C : 1 \rightarrowtail U_0}{\Delta \dashv C \circ [] \triangleleft () : \lfloor_n 1 \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillL}\ \frac{\Delta \dashv C : \lfloor_n T_1 \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ [] \triangleleft \mathsf{Inl} : \lfloor_n T_1 \oplus T_2 \rfloor \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-FillR}\ \frac{\Delta \dashv C : \lfloor_n T_2 \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ [] \triangleleft \mathsf{Inr} : \lfloor_n T_1 \oplus T_2 \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillP}\ \frac{\Delta \dashv C : (\lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor) \rightarrowtail U_0}{\Delta \dashv C \circ [] \triangleleft (,) : \lfloor_n T_1 \otimes T_2 \rfloor \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-FillE}\ \frac{\Delta \dashv C : \lfloor_{m\cdot n} T \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ [] \triangleleft \mathsf{E}_m : \lfloor_n !_m T \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillF}\ \frac{\Delta_1,\ (1\uparrow{\cdot}n){\cdot}\Delta_2 \dashv C : 1 \rightarrowtail U_0 \quad \Delta_2,\ x :_m T \vdash u : U}{\Delta_1 \dashv C \circ [] \triangleleft (\lambda x\, {}_m\!\mapsto u) : \lfloor_n T \to_m U \rfloor \rightarrowtail U_0} \qquad \textsc{Ty-ectxs-FillComp1}\ \frac{\Delta_1,\ (1\uparrow{\cdot}n){\cdot}\Delta_2 \dashv C : T \rightarrowtail U_0 \quad \Delta_2 \vdash t' : U \ltimes T}{\Delta_1 \dashv C \circ [] \lessdot\!\circ t' : \lfloor_n U \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillComp2}\ \frac{\Delta_1,\ (1\uparrow{\cdot}n){\cdot}\Delta_2 \dashv C : T \rightarrowtail U_0 \quad \Delta_1 \vdash v : \lfloor_n U \rfloor}{\Delta_2 \dashv C \circ v \lessdot\!\circ [] : U \ltimes T \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-OpenAmpar}\ \frac{\mathtt{LinOnly}\ \Delta_3 \quad \mathtt{FinAgeOnly}\ \Delta_3 \quad \mathbf{hnames}(C)\ \#\#\ \mathbf{hnames}(\Delta_3) \quad \Delta_1,\ \Delta_2 \dashv C : (U \ltimes T') \rightarrowtail U_0 \quad \Delta_2,\ \rightarrow^{-1}\Delta_3 \Vdash v_2 : U}{1\uparrow{\cdot}\Delta_1,\ \Delta_3 \dashv C \circ {}^{\mathrm{op}}_{\mathbf{hnames}(\Delta_3)}\langle v_2 \wedge [] \rangle : T' \rightarrowtail U_0}$$

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

$$\textsc{Ty-cmd}\ \frac{\Delta \dashv C : T \rightarrowtail U_0 \quad \Delta \vdash t : T}{\vdash C[t] : U_0}$$

Fig. 11. Typing rules for evaluation contexts and commands

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).

6.3 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 C;
- unfocus rules, where the term under focus is in fact a value, and thus we pop a focusing component from the stack C 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:

$$\mathsf{C}\big[\mathbf{case}_{\mathsf{m}}\ \mathsf{t}\ \mathbf{of}\ (x_1\,,x_2)\mapsto \mathsf{u}\big] \longrightarrow \big(\mathsf{C}\circ \mathbf{case}_{\mathsf{m}}\ []\ \mathbf{of}\ (x_1\,,x_2)\mapsto \mathsf{u}\big)\big[\mathsf{t}\big] \quad \textit{when}\quad \texttt{NotVal t}$$

$$\big(\mathsf{C}\circ \mathbf{case}_{\mathsf{m}}\ []\ \mathbf{of}\ (x_1\,,x_2)\mapsto \mathsf{u}\big)\big[\mathsf{v}\big] \longrightarrow \mathsf{C}\big[\mathbf{case}_{\mathsf{m}}\ \mathsf{v}\ \mathbf{of}\ (x_1\,,x_2)\mapsto \mathsf{u}\big]$$

$$\mathsf{C}\big[\mathbf{case}_{\mathsf{m}}\ (\mathsf{v}_1\,,\mathsf{v}_2)\ \mathbf{of}\ (x_1\,,x_2)\mapsto \mathsf{u}\big] \longrightarrow \mathsf{C}\big[\mathsf{u}[x_1\coloneqq \mathsf{v}_1][x_2\coloneqq \mathsf{v}_2]\big]$$

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 12, but the whole system is included in the annex (see Figures 13 and 14).

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 $\mathsf{C}\langle\!|h\coloneqq_H \mathsf{v}|\!\rangle$ on the evaluation context C. 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 $\mathsf{C}\langle\!|h\coloneqq_H \mathsf{v}|\!\rangle$ 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 $\mathbf{hnames}(\mathsf{C})\ \#\#\ \mathbf{hnames}(\Delta_3)$ 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.

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

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

We extend $\bullet[H^{\pm}h']$ to arbitrary values, extended terms, and typing contexts in the obvious way (keeping in mind that ${}_{H'}\langle \mathsf{v}_2\,{}_{\wedge}\,\mathsf{v}_1\rangle$ 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

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

$\mathsf{C}[(^{\mathsf{v}}\lambda x\ {}_{\mathsf{m}}\!\mapsto \mathsf{u})\ \mathsf{v}] \longrightarrow \mathsf{C}[\mathsf{u}[x := \mathsf{v}]]$ RED-APP

$\mathsf{C}[() \fatsemi \mathsf{u}] \longrightarrow \mathsf{C}[\mathsf{u}]$ RED-PATU

$\mathsf{C}[\mathbf{case}_{\mathsf{m}}\ (\mathsf{Inl}\ \mathsf{v}_1)\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathsf{u}_1,\ \mathsf{Inr}\ x_2 \mapsto \mathsf{u}_2\}] \longrightarrow \mathsf{C}[\mathsf{u}_1[x_1 := \mathsf{v}_1]]$ RED-PATL

$\mathsf{C}[\mathbf{case}_{\mathsf{m}}\ (\mathsf{Inr}\ \mathsf{v}_2)\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathsf{u}_1,\ \mathsf{Inr}\ x_2 \mapsto \mathsf{u}_2\}] \longrightarrow \mathsf{C}[\mathsf{u}_2[x_2 := \mathsf{v}_2]]$ RED-PATR

$\mathsf{C}[\mathbf{case}_{\mathsf{m}}\ (\mathsf{v}_1, \mathsf{v}_2)\ \mathbf{of}\ (x_1, x_2) \mapsto \mathsf{u}] \longrightarrow \mathsf{C}[\mathsf{u}[x_1 := \mathsf{v}_1][x_2 := \mathsf{v}_2]]$ RED-PATP

$\mathsf{C}[\mathbf{case}_{\mathsf{m}}\ \mathsf{E}_{\mathsf{n}}\ \mathsf{v}'\ \mathbf{of}\ \mathsf{E}_{\mathsf{n}}\ x \mapsto \mathsf{u}] \longrightarrow \mathsf{C}[\mathsf{u}[x := \mathsf{v}']]$ RED-PATE

$\mathsf{C}[\mathbf{to}_{\ltimes}\ \mathsf{v}_2] \longrightarrow \mathsf{C}[{}_{\{\}}\langle \mathsf{v}_2 \wedge () \rangle]$ RED-TOA

$\mathsf{C}[\mathbf{from}_{\ltimes}\ {}_{\{\}}\langle \mathsf{v}_2 \wedge \mathsf{E}_{1\infty}\ \mathsf{v}_1 \rangle] \longrightarrow \mathsf{C}[(\mathsf{v}_2, \mathsf{E}_{1\infty}\ \mathsf{v}_1)]$ RED-FROMA

$\mathsf{C}[\mathsf{alloc}] \longrightarrow \mathsf{C}[{}_{\{1\}}\langle \boxed{1} \wedge \to 1 \rangle]$ RED-ALLOC

$\mathsf{C}[\to h \triangleleft ()] \longrightarrow \mathsf{C}(\!|h :=_{\{\}}\ ()|\!)[()]$ RED-FILLU

$\mathsf{C}[\to h \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\mapsto \mathsf{u})] \longrightarrow \mathsf{C}(\!|h :=_{\{\}}\ {}^{\mathsf{v}}\lambda x\ {}_{\mathsf{m}}\!\mapsto \mathsf{u}|\!)[()]$ RED-FILLF

$\mathsf{C}[\to h \triangleleft \mathsf{Inl}] \longrightarrow \mathsf{C}(\!|h :=_{\{h'+1\}}\ \mathsf{Inl}\ \boxed{h'+1}|\!)[\to h'+1]$ RED-FILLL

$\mathsf{C}[\to h \triangleleft \mathsf{Inr}] \longrightarrow \mathsf{C}(\!|h :=_{\{h'+1\}}\ \mathsf{Inr}\ \boxed{h'+1}|\!)[\to h'+1]$ RED-FILLR

$\mathsf{C}[\to h \triangleleft \mathsf{E}_{\mathsf{m}}] \longrightarrow \mathsf{C}(\!|h :=_{\{h'+1\}}\ \mathsf{E}_{\mathsf{m}}\ \boxed{h'+1}|\!)[\to h'+1]$ RED-FILLE

$\mathsf{C}[\to h \triangleleft (,)] \longrightarrow \mathsf{C}(\!|h :=_{\{h'+1, h'+2\}}\ (\boxed{h'+1}, \boxed{h'+2})|\!)[(\to h'+1, \to h'+2)]$ RED-FILLP

$\mathsf{C}[\to h \mathrel{\text{◂◂}}_H \langle \mathsf{v}_2 \wedge \mathsf{v}_1 \rangle] \longrightarrow \mathsf{C}(\!|h :=_{(H \pm h')}\ \mathsf{v}_2[H \pm h']|\!)[\mathsf{v}_1[H \pm h']]$ RED-FILLCOMP

$\mathsf{C}[\mathbf{map}_H \langle \mathsf{v}_2 \wedge \mathsf{v}_1 \rangle\ \mathbf{with}\ x \mapsto \mathsf{t}'] \longrightarrow (\mathsf{C} \circ {}^{\mathsf{op}}_{H \pm h''}\langle \mathsf{v}_2[H \pm h''] \wedge [] \rangle)[\mathsf{t}'[x := \mathsf{v}_1[H \pm h'']]]$ OPEN-AMPAR

$(\mathsf{C} \circ {}^{\mathsf{op}}_{H}\langle \mathsf{v}_2 \wedge [] \rangle)[\mathsf{v}_1] \longrightarrow \mathsf{C}[{}_{H}\langle \mathsf{v}_2 \wedge \mathsf{v}_1 \rangle]$ CLOSE-AMPAR

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

Fig. 12. Small-step semantics

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 ${}^{\mathsf{op}}_{H \pm h'}\langle \mathsf{v}_2[H \pm h'] \wedge [] \rangle$ 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 C 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 ${}_H\langle \mathsf{v}_2 \wedge \mathsf{v}_1 \rangle$ to a hole $\boxed{h}$ that is part of a structure with holes somewhere inside C. 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 $\mathsf{v}_2[H \pm h'']$ and $\mathsf{v}_1[H \pm h'']$.

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

THEOREM 6.1 (TYPE PRESERVATION). *If* $\vdash \mathsf{C}[\mathsf{t}] : \mathsf{T}$ *and* $\mathsf{C}[\mathsf{t}] \longrightarrow \mathsf{C}'[\mathsf{t}']$ *then* $\vdash \mathsf{C}'[\mathsf{t}'] : \mathsf{T}$.

THEOREM 6.2 (PROGRESS). *If* $\vdash \mathsf{C}[\mathsf{t}] : \mathsf{T}$ *and* $\forall \mathsf{v}, \mathsf{C}[\mathsf{t}] \neq [][\mathsf{v}]$ *then* $\exists \mathsf{C}', \mathsf{t}'.\ \mathsf{C}[\mathsf{t}] \longrightarrow \mathsf{C}'[\mathsf{t}']$.

A command of the form $[]\left[v\right]$ 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.

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 [Sewell et al. 2007], the same Ott file is used as a source for this article, making sure that we've proved the same system as we're presenting; though some visual simplification is applied by a script to produce the version in the article.

Most of the proof was done by 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.

8 IMPLEMENTATION OF λ_d USING IN-PLACE MEMORY MUTATIONS

The formal language presented in Sections 5 and 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. $\mathsf{E}_{\omega\nu}\ \{h\}\langle 0 :: \boxed{h}_{\wedge} \to h\rangle$ (representing a non-linear difference list $0::\square$), and then used twice, each time in a different way:

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

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 6.3, every time an ampar is **map**ped 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 2. Copy-on-write can be optimized using fully-in-place functional programming in the style of [Lorenzen et al. 2023], 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 [Spiwack et al. 2022] and introduce a new type Token, together with primitives **dup** and **drop** (remember that unqualified arrows have mode 1ν, so are linear):

$$\begin{array}{l}\textbf{dup}\ :\ \mathsf{Token} \to \mathsf{Token} \otimes \mathsf{Token} \\ \textbf{drop}\ :\ \mathsf{Token} \to 1 \\ \textbf{alloc}_{\textbf{ip}}\ :\ \mathsf{Token} \to \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor \\ \mathsf{alloc}_{\mathsf{cow}}\ :\ \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor\end{array}$$

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 $\textbf{alloc}_{\textbf{ip}}$ have a linear dependency on a linear tok_k variable. If an ampar produced by $\textbf{alloc}_{\textbf{ip}}\ tok_k$ were to be used twice in a block t, then t would require a typing context $\{tok_k :_{\omega\nu} \mathsf{Token}\}$, 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 $\{tok_0 :_{1\infty} \mathsf{Token}$ is a slightly more ergonomic solution than adding a primitive function $\textbf{withToken}\ :\ (\mathsf{Token} \to_{1\infty} !_{\omega\infty}\mathsf{T}) \to !_{\omega\infty}\mathsf{T}$ as it is done in [Bagrel 2024].

In λ_d with Tokens and $\textbf{alloc}_{\textbf{ip}}$, 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 ◃◦ 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.

9 RELATED WORK

9.1 Destination-passing style for efficient memory management

In [Shaikhha et al. 2017], 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.

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.

9.2 Tail modulo constructor

Another example of destinations in a compiler's optimizer is [Bour et al. 2021]. It's meant to address the perennial problem that the map function on linked lists isn't tail-recursive, hence consumes stack space. The observation is that there's a systematic transformation of functions where the only recursive call is under a constructor to a destination-passing tail-recursive implementation.

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

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

to ask the compiler to perform the translation. The compiler will then throw an error if it can't. This way, contrary to the optimizations in [Shaikhha et al. 2017], 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.

9.3 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 [Minamide 1998]. 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 $S \ltimes T$, only the right hand part.

In [Minamide 1998], it's only ever possible to represent structures with a single hole. But this is a rather superficial restriction. The author doesn't comment on this, but we believe that this

restriction only exists for convenience of the exposition: the language is lowered to a language without function abstraction and where composition is performed by combinators. While it's easy to write a combinator for single-argument-function composition, it's cumbersome to write combinators for functions with multiple arguments. But having multiple-hole data structures wouldn't have changed their system in any profound way.

The more important difference is that while their system is based on a type of linear functions, 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 [Minamide 1998] doesn't have. This means that using [Minamide 1998] — or the more recent but similarly expressive system from [Lorenzen et al. 2024] — one can implement the examples with difference lists and queues from Section 2.2, but they can't do our breadth-first traversal example from Section 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 [Minamide 1998] 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.

9.4 Destination-passing style programming: a Haskell implementation

In [Bagrel 2024], 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 [Minamide 1998].

In their system, $d \blacktriangleleft \mathsf{t}$ requires t to be unrestricted, while in λ_d, t can be linear. One consequence is that in [Bagrel 2024], 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 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, [Bagrel 2024] 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.

9.5 Semi-axiomatic sequent calculus

In [DeYoung et al. 2020], 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.

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.

REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (aug 2020), 28 pages. https://doi.org/10.1145/3408972

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

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

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–29. https://doi.org/10.1145/3158093 arXiv:1710.09756 [cs].

Frédéric Bour, Basile Clément, and Gabriel Scherer. 2021. Tail Modulo Cons. In *JFLA 2021 - Journées Francophones des Langages Applicatifs*. Saint Médard d'Excideuil, France. https://inria.hal.science/hal-03146495

Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/351240.351262

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. 2020. Semi-Axiomatic Sequent Calculus. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:22. https://doi.org/10.4230/LIPIcs.FSCD.2020.29

Jeremy Gibbons. 1993. Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips. No. 71 (1993). https://www.cs.ox.ac.uk/publications/publication2363-abstract.html Number: No. 71.

Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2023. Phases in Software Architecture. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture*. ACM, Seattle WA USA, 29–33. https://doi.org/10.1145/3609025.3609479

J.-Y. Girard. 1995. Linear Logic: its syntax and semantics. In *Advances in Linear Logic*, Jean-Yves Girard, Yves Lafont, and Laurent Regnier (Eds.). Cambridge University Press, Cambridge, 1–42. https://doi.org/10.1017/CBO9780511629150.002

Robert Hood and Robert Melville. 1981. Real-time queue operations in pure LISP. *Inform. Process. Lett.* 13, 2 (1981), 50–54. https://doi.org/10.1016/0020-0190(81)90030-2

John Hughes. 1986. A Novel Representation of Lists and its Application to the Function "reverse". *Inf. Process. Lett.* 22 (01 1986), 141–144.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 275–304. https://doi.org/10.1145/3607840

Anton Lorenzen, Daan Leijen, Wouter Swierstra, and Sam Lindley. 2024. The Functional Essence of Imperative Binary Search Trees. *Proc. ACM Program. Lang.* 8, PLDI, Article 168 (jun 2024), 25 pages. https://doi.org/10.1145/3656398

Yasuhiko Minamide. 1998. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/268946.268953

Chris Okasaki. 2000. Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 131–136. https://doi.org/10.1145/351240.351253

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (jul 2019), 30 pages. https://doi.org/10.1145/3341714

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1291151.1291155

Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. ACM, Oxford UK, 12–23. https://doi.org/10.1145/3122948.3122949

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly qualified types: generic inference for capabilities and uniqueness. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 95:137–95:164. https://doi.org/10.1145/3547626