# Destination calculus

A linear $\lambda$–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 similar 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 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 $\lambda$–calculus for purely functional memory writes. *Proc. ACM Program. Lang.* XX, XX, Article XXX (January 2025), 27 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 $T \to U$ would, in destination-passing style, have type $T \to \lfloor U \rfloor \to 1$ instead. This style is common in system programming, where destinations $\lfloor U \rfloor$ are more commonly known as "out parameters". In C, $\lfloor U \rfloor$ would typically be a pointer of type $U*$.

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].

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 programming 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 $\lambda_d$. We intend $\lambda_d$ to serve as a core calculus to reason about safe destinations. Indeed $\lambda_d$ subsumes all the systems that we've discussed in this section: they can all be encoded in $\lambda_d$ 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 $\lambda_d$.

Our contributions are as follows:

- $\lambda_d$, a linear and modal simply typed $\lambda$-calculus with destinations (Sections 5 and 6). $\lambda_d$ is expressive enough so that previous calculi for destinations can be encoded in $\lambda_d$ (see Section 9);
- a demonstration that $\lambda_d$ 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 $\lambda_d$ which uses mutation without compromising the purity of $\lambda_d$ (Section 8);
- formally-verified proofs, with the Coq proof assistant, of type safety (Section 7).

## 2 WORKING WITH DESTINATIONS

Let's introduce and get familiar with $\lambda_d$, our simply typed $\lambda$-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 $\lambda_d$ 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 $\lambda_d$ you would have $\mathsf{T} \to \lfloor \mathsf{U} \rfloor \to 1$, where $\lfloor \mathsf{U} \rfloor$ is read "destination for type $\mathsf{U}$". For instance, here is a destination-passing version of the identity function:

$$\begin{array}{l} \mathsf{dId} \;:\; \mathsf{T} \to \lfloor \mathsf{T} \rfloor \to 1 \\ \mathsf{dId}\ x\ d \;\triangleq\; d \blacktriangleleft x \end{array}$$

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

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

$$\begin{aligned}&\mathbf{fillWithInlCtor} \;:\; \lfloor \mathsf{T} \oplus \mathsf{U} \rfloor \to \lfloor \mathsf{T} \rfloor\\ &\mathbf{fillWithInlCtor}\; d \;\triangleq\; d \triangleleft \mathsf{Inl}\end{aligned}$$

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

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

$$\begin{aligned}&\mathbf{fillWithPairCtor} \;:\; \lfloor \mathsf{T} \otimes \mathsf{U} \rfloor \to \lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor\\ &\mathbf{fillWithPairCtor}\; d \;\triangleq\; d \triangleleft (,)\end{aligned}$$

After using **fillWithPairCtor**, the user must fill both the first field *and* the second field, using the destinations of type $\lfloor \mathsf{T} \rfloor$ and $\lfloor \mathsf{U} \rfloor$ 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 $\lfloor \mathsf{T} \rfloor$ is a pointer to a yet-undefined part of a bigger structure. We say that such a structure has a hole of type $\mathsf{T}$; 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 $\mathsf{T} \otimes \mathsf{U}$ 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, $\lambda_d$ introduces a type $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ for a structure of type $\mathsf{S}$ missing a value of type $\mathsf{T}$ to be complete. There can be several holes in $\mathsf{S}$, resulting in several destinations on the right hand side: for example, $\mathsf{S} \ltimes (\lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor)$ represents a $\mathsf{S}$ that misses both a $\mathsf{T}$ and a $\mathsf{U}$ to be complete.

The general form $\mathsf{S} \ltimes \mathsf{T}$ is read "$\mathsf{S}$ ampar $\mathsf{T}$". 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 $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ is akin to a "par" type $\mathsf{S} \parr \mathsf{T}^{\perp}$ in linear logic; you can think of $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ as a (linear) function from $\mathsf{T}$ to $\mathsf{S}$. That structures with holes could be seen as 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, $\lambda_d$ 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:

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

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

To tie this up, we need a way to introduce and to eliminate structures with holes. Structures with holes are introduced with **alloc**[1] which creates a value of type $\mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor$. **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[2] $\mathbf{from}'_{\ltimes} : \mathsf{S} \ltimes 1 \to \mathsf{S}$: 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, $\lambda_d$ doesn't have primitive constructor forms, constructors in $\lambda_d$ are syntactic sugar. We show here the definition of Inl and (,), but the other constructors are derived similarly.

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

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

*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 $\lambda_d$ uses a linear type system to ensure that:

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

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

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

  $$\begin{aligned}&\mathsf{ambiguous1} : \mathsf{Bool}\\ &\mathsf{ambiguous1} \triangleq \mathbf{from}'_{\ltimes}(\mathbf{map}\ \mathbf{alloc}\ \mathbf{with}\ d \mapsto d \blacktriangleleft \mathsf{true} \mathbin{⨾} d \blacktriangleleft \mathsf{false})\end{aligned}$$

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

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

[1] Despite its name, **alloc** isn't the only source of memory allocations in the language: filling primitives like ◃Inl also have to allocate space for the corresponding hollow constructor.
[2] As the name suggest, there is a more general elimination $\mathbf{from}_{\ltimes}$. It will be discussed in Section 5.

```
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′
to_List : DList T → List T
to_List ys ≜ from′_⋉ (map ys with d ↦ d ◂ [])
```

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

Fig. 1. Difference list and queue implementation in equirecursive $\lambda_d$

## 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 this section, we suppose that $\lambda_d$ has equirecursive types and a fixed-point operator. These aren't part of the formal system of Section 5 but don't add any complication.

*Linked lists.* We define lists as a fixpoint, as usual: $\mathsf{List\ T} \overset{\text{rec}}{\triangleq} 1\oplus(\mathsf{T}\otimes(\mathsf{List\ T}))$. For convenience, we also define filling operators ◂[] and ◂(::):

```
◂[] : ⌊List T⌋ → 1
d ◂ [] ≜ d ◂ Inl ◂ ()
```

```
◂(::) : ⌊List T⌋ → ⌊T⌋⊗⌊List T⌋
d ◂ (::) ≜ d ◂ Inr ◂ (,)
```

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

```
(::) : T⊗(List T) → List T
x :: xs ≜ from′_⋉ (map alloc with d ↦ case (d ◂ (::)) of (dx, dxs) ↦ dx ◀ x ⨾ dxs ◀ xs)
```

*Difference lists.* Just like in any language, iterated concatenation of lists $((xs_1 +\!\!+ xs_2) +\!\!+ \ldots) +\!\!+ xs_n$ is quadratic in $\lambda_d$. The usual solution to this is difference lists. The name difference lists covers many related implementations, 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. A difference list is 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. In $\lambda_d$, we can do better by representing a difference list as a list with a hole. A singleton difference list is $x$::□. Concatenation is filling the hole with another difference list, using operator ◂∘. The details are on the left of Figure 1. The $\lambda_d$ encoding for difference lists makes no superfluous traversal: concatenation is just an $O(1)$ in-place update.

*Efficient queue using difference lists.* In an immutable functional language, a queue can be implemented as a pair of lists (*front*, *back*) [Hood and Melville 1981]. *back* stores new elements in reverse order ($O(1)$ prepend). We pop elements from *front*, except when it is empty, in which case we set the queue to (**reverse** *back*, []), and pop from the new front.

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

So we can give an improved version of the simple functional queue using destinations. This implementation is presented on the right-hand side of Figure 1. Note that contrary to an imperative programming language, we can't implement the queue as a single difference list: our type system prevents us from reading the front elements of difference lists. Just like for the simple functional queue, we need a pair of one list that we can read from, and one that we can extend. Nevertheless this implementation of queues is both pure, as guaranteed by the $\lambda_d$ type system, and nearly as efficient as what an imperative programming language would afford.

## 3 SCOPE ESCAPE OF DESTINATIONS

In Section 2, we've been making an implicit assumption: establishing a linear discipline on destinations ensures that all destinations will eventually find their way to the left of a fill operator $\blacktriangleleft$ or $\triangleleft$, so that the associated holes get written to. This turns out to be slightly incomplete.

To see why, let's consider the type $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$: 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 place in the structure. For instance, if we have $d : \lfloor \mathsf{T} \rfloor$ pointing to hole $\boxed{h}$ in structure $\mathsf{v}$ and $dd : \lfloor\lfloor \mathsf{T} \rfloor\rfloor$ pointing to hole $\boxed{hd}$ in structure $\mathsf{vd}$, we can form $dd \blacktriangleleft d$: destination $d$ will be stored inside $\mathsf{vd}$.

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

- If we count this as a non-linear use of $d$, then $dd \blacktriangleleft d$ is rejected since destinations (here $d$) can only be used linearly. This is certainly possible, and this is, in fact what [Bagrel 2024] does. However, this is a blunt limitation, and it would prevent us from storing destinations in structures with holes, as we do, crucially, in Section 4.
- If we do not count this use of $d$ at all, we can write $dd \blacktriangleleft d \fatsemi d \blacktriangleleft \mathsf{v}$ in order to write to the hole $\boxed{h}$ twice, which is unsound, as discussed in Section 2.1.

So linear use it is. But it creates a problem: there's no way, within the linear type system, to distinguish between "the hole $\boxed{h}$ has been filled" and "$d$ has been stored (and $\boxed{h}$ still exists)". Depending on the nesting of $\mathsf{v}$ and $\mathsf{vd}$, this may allow us to read uninitialized memory $\boxed{h}$.

Let's compare two examples. We assume a simple store semantics[3] where structures with holes stay in the store until they are completed. We'll need the $\mathbf{alloc'} : (\lfloor \mathsf{T} \rfloor \rightarrow 1) \rightarrow \mathsf{T}$ operator. The semantics of $\mathbf{alloc'}$ is: allocate a structure with a single root hole in the store, call the user-supplied function with the destination to the root hole as an argument; when the function has consumed all destinations (so only unit remains), pop the structure from the store to obtain a complete $\mathsf{T}$.

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

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

[3] Our actual semantics (Section 6) isn't a store semantics. It enforces invariants syntactically, making the proofs much easier.

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

$$\begin{array}{rr}
 & \{\} \mid \mathbf{alloc}'\ (\lambda dd \mapsto \mathbf{case}\ (\mathbf{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d))\ \mathbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\}) \\
\longrightarrow & \{\mathsf{vd} := \boxed{hd}\} \mid \mathbf{case}\ (\mathbf{alloc}'\ (\lambda d \mapsto {\rightarrow}hd \blacktriangleleft d))\ \mathbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd} \\
\longrightarrow & \{\mathsf{vd} := \boxed{hd}, \mathsf{v} := \boxed{h}\} \mid \mathbf{case}\ ({\rightarrow}hd \blacktriangleleft {\rightarrow}h \fatsemi \mathbf{deref}\ \mathsf{v})\ \mathbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd} \\
\longrightarrow & \{\mathsf{vd} := {\rightarrow}h, \underline{\mathsf{v} := \boxed{h}}\} \mid \mathbf{case}\ (\underline{\mathbf{deref}\ \mathsf{v}})\ \mathbf{of}\ \{\mathsf{true} \mapsto (),\ \mathsf{false} \mapsto ()\} \fatsemi \mathbf{deref}\ \mathsf{vd}
\end{array}$$

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

Again, using purely a linear type system, we can only reject this example if we also reject the first, sound example, as in [Bagrel 2024]. In this case, the type $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$ becomes practically useless: such destinations can never be filled.

This isn't the direction we want to take: we really want to be able to store destinations in data structures with holes. So we want t in $d \blacktriangleleft$ t to be allowed to be linear. Without further restrictions, it wouldn't be sound, so to address this, $\lambda_d$ 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 $\lambda_d$, 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 so it requires fancy workarounds [Gibbons 1993; Gibbons et al. 2023; Okasaki 2000].

On the other hand, first-class destination passing lets us use the familiar imperative algorithm for breadth-first traversal where a queue drives the processing order. When the element (*input subtree*, *destination to output subtree*) at the front of the queue has been processed, the children nodes and destinations are enqueued to be processed later.

Figure 2 presents the $\lambda_d$ implementation of the breadth-first tree traversal. There, Tree T is defined unsurprisingly as Tree T $\triangleq$ $1 \oplus (\mathsf{T} \otimes ((\mathsf{Tree\ T}) \otimes (\mathsf{Tree\ T})))$; we refer to the constructors of Tree T as Nil and Node, defined in the obvious way. We also assume some encoding of the type Nat of natural number. Queue T is the efficient queue type from Section 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.

In **mapAccumBFS**, we create a new destination *dtree* into which we will write the result of the traversal, then call **go**. The **go** function is in destination-passing style, but what's remarkable is that **go** takes an unbounded number of destinations as arguments, since there are as many destinations as items in the queue. This is where we use the fact that destinations are ordinary values.

The implementation of Figure 2 is very close to the one 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.

However this improvement comes with a cost: we a *mode* system that combine linearity and age to make the system sound, hence the new fuchsia annotations in the code. We'll describe modes in

```
go : (S ω∞⟶ T₁ ⟶ (!ω∞ S)⊗T₂) ω∞⟶ S ω∞⟶ Queue (Tree T₁⊗⌊Tree T₂⌋) ⟶ 1
         rec
go f st q  ≜  case (dequeue q) of {
                  Inl () ↦ (),
                  Inr ((tree, dtree), q′) ↦ case tree of {
                      Nil ↦ dtree ◃ Nil ⨟ go f st q′,
                      Node x tl tr ↦ case (dtree ◃ Node) of
                          (dy, (dtl, dtr)) ↦ case (f st x) of
                              (Eω∞ 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 ⨟ (Eω∞ (succ st), st)) 1 tree
```

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

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

Fig. 3. Grammar of $\lambda_d$

detail in Section 5. In the meantime, $1$ and $\omega$ control linearity: we use $\omega$ to mean that the state and function $f$ can be used many times. On the other hand, $\infty$ is an *age* annotation; in particular, the associated argument cannot carry destinations. Arguments with no modes are otherwise linear and can capture destinations. We introduce the exponential modality $!_{\mathsf{m}}\mathsf{T}$ to reify mode $\mathsf{m}$ 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 $(!_{\mathsf{m}}\mathsf{T}) \to \mathsf{U} \simeq \mathsf{T}\ {}_{\mathsf{m}}\!\!\to \mathsf{U}$.

## 5 TYPE SYSTEM

$\lambda_d$ is a simply typed $\lambda$-calculus with unit ($1$), product ($\otimes$) and sum ($\oplus$) types. Its most salient features are the destination $\lfloor_{\mathsf{n}}\mathsf{T}\rfloor$ and ampar $\mathsf{S} \ltimes \mathsf{T}$ types which we've introduced in Sections 2 to 4. Just as important are *modes* and the associated exponential modality $!_{\mathsf{m}}$. The grammar of $\lambda_d$ 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.

$$
\begin{aligned}
\mathsf{Inl}\ \mathrm{t} &\triangleq \mathsf{from}'_{\ltimes}\,(\mathsf{map}\ \mathsf{alloc}\ \mathsf{with}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft \mathrm{t}) \\
\mathsf{Inr}\ \mathrm{t} &\triangleq \mathsf{from}'_{\ltimes}\,(\mathsf{map}\ \mathsf{alloc}\ \mathsf{with}\ d \mapsto d \triangleleft \mathsf{Inr} \blacktriangleleft \mathrm{t}) \\
\mathsf{E}_{\mathrm{m}}\ \mathrm{t} &\triangleq \mathsf{from}'_{\ltimes}\,(\mathsf{map}\ \mathsf{alloc}\ \mathsf{with}\ d \mapsto d \triangleleft \mathsf{E}_{\mathrm{m}} \blacktriangleleft \mathrm{t}) \\
\lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u} &\triangleq \mathsf{from}'_{\ltimes}\,(\mathsf{map}\ \mathsf{alloc}\ \mathsf{with}\ d \mapsto d \triangleleft (\lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{from}'_{\ltimes}\ \mathrm{t} \triangleq \\
&\quad \mathsf{case}\ (\mathsf{from}_{\ltimes}\ (\mathsf{map}\ \mathrm{t}\ \mathsf{with}\ un \mapsto un \fatsemi \mathsf{E}_{1\infty}\ ()))\ \mathsf{of} \\
&\qquad (st\,, ex) \mapsto \mathsf{case}\ ex\ \mathsf{of} \\
&\qquad\quad \mathsf{E}_{1\infty}\ un \mapsto un \fatsemi st \\
&() \triangleq \mathsf{from}'_{\ltimes}\,(\mathsf{map}\ \mathsf{alloc}\ \mathsf{with}\ d \mapsto d \triangleleft ()) \\
&(\mathrm{t}_1\,, \mathrm{t}_2) \triangleq \mathsf{from}'_{\ltimes}\,(\mathsf{map}\ \mathsf{alloc}\ \mathsf{with}\ d \mapsto \mathsf{case} \\
&\qquad (d \triangleleft (,))\ \mathsf{of} \\
&\qquad\quad (d_1\,, d_2) \mapsto d_1 \blacktriangleleft \mathrm{t}_1 \fatsemi d_2 \blacktriangleleft \mathrm{t}_2)
\end{aligned}
$$

Fig. 4. Syntactic sugar for terms

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

| $+$ | $\uparrow^k$ | $\infty$ |
|---|---|---|
| $\uparrow^j$ | if $k = j$ then $\uparrow^k$ else $\infty$ | $\infty$ |
| $\infty$ | $\infty$ | $\infty$ |

| $\cdot$ | $\uparrow^k$ | $\infty$ |
|---|---|---|
| $\uparrow^j$ | $\uparrow^{k+j}$ | $\infty$ |
| $\infty$ | $\infty$ | $\infty$ |

| $+$ | $1$ | $\omega$ |
|---|---|---|
| $1$ | $\omega$ | $\omega$ |
| $\omega$ | $\omega$ | $\omega$ |

| $\cdot$ | $1$ | $\omega$ |
|---|---|---|
| $1$ | $1$ | $\infty$ |
| $\omega$ | $\infty$ | $\infty$ |

Fig. 5. Operation tables for age and multiplicity semirings

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[4]. In $\lambda_d$ 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 $1\nu$ of the semiring. In particular, a function arrow without annotation, or with multiplicity $1$, is linear; it is equivalent to $\multimap$ from linear logic [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 **map** t **with** $x \mapsto \mathrm{t}'$. If we have a term **map** $\mathrm{t}_1$ **with** $x_1 \mapsto$ **map** $\mathrm{t}_2$ **with** $x_2 \mapsto$ **map** $\mathrm{t}_3$ **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 $\infty$ for values which don't originate from the scope of a **map** t **with** $x \mapsto \mathrm{t}'$ and can be freely used in and returned by any scope. In particular, destinations can never have age $\infty$; the main role of age $\infty$ 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 one.

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.

[4]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].

## 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 $(\mathsf{T}, \mathsf{S})\mathsf{hfun}$ is thus a weak form of linear lambda abstraction $\mathsf{T} \multimap \mathsf{S}$, which just moves a piece of data into a bigger data structure.

Now, in classical linear logic (CLL), we can transform linear implication $\mathsf{T} \multimap \mathsf{S}$ into $\mathsf{S} \parr \mathsf{T}^{\perp}$. Doing so for the type $(\mathsf{T}, \mathsf{S})\mathsf{hfun}$ gives $\mathsf{S} \parr \lfloor \mathsf{T} \rfloor$, where $\lfloor \cdot \rfloor$ is a form of dualisation. Notation is slightly abusive here; this isn't exactly the same $\parr$ as in CLL, because $\mathsf{hfun}$ is not as powerful as $\multimap$.

Transforming the hole abstraction from its original implication form to a *par* form lets us consider the dualized type $\lfloor \mathsf{T} \rfloor$ — 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* $\ltimes$. In CLL any of the sides $\mathsf{S}$ or $\mathsf{T}$ of a par $\mathsf{S} \parr \mathsf{T}$ can be eliminated, by interaction with the dual type $\cdot^{\perp}$, which then frees up the other side. But in $\lambda_d$, we have two types of interaction to consider: interaction between $\mathsf{T}$ and destination $\lfloor \mathsf{T} \rfloor$, and interaction between $\mathsf{T}$ and functions $\mathsf{T} \to \cdot$. The structure containing holes, $\mathsf{S}$, can safely interact with $\lfloor \mathsf{S} \rfloor$ (merge it into another structure with holes), but not with $\mathsf{S} \to \cdot$, as it would let us read an incomplete structure!

On the other hand, a complete value of type $\mathsf{T} = (\ldots \lfloor \mathsf{T}' \rfloor \ldots)$ containing destinations (but no holes) can safely interact with a function $f : \mathsf{T} \to 1$: in particular, $f$ can pattern-match on the value of type $\mathsf{T}$ 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 $\mathsf{S}$ can contain holes, and can be only be eliminated by composition with another structure having a destination $\lfloor \mathsf{S} \rfloor$ using operator $\lhd\!\circ$ (the right side $\mathsf{T}$ is then returned);
- the right side $\mathsf{T}$ cannot contain holes (it might contain destinations), and can be eliminated by interaction with $\mathsf{T} \to 1$ to free up the left side $\mathsf{S}$. This is done using $\mathbf{from}'_{\ltimes}$ and $\mathbf{map}$.

## 5.3 Typing rules

The typing rules for $\lambda_d$ 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 $\texttt{DisposableOnly}\ \Gamma$ in rules Ty-term-Var, Ty-term-Alloc and Ty-sterm-Unit says that $\Gamma$ can only contain bindings with multiplicity $\omega$, 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 $1\nu <: m$ defined as $pa <: p'a' \iff p <:^{p} p' \land a <:^{a} a'$ where:

$$1 <:^{p} 1 \qquad p <:^{p} \omega \qquad a <:^{a} \infty \qquad \uparrow^{k} <:^{a} \uparrow^{j} \iff k = j \quad \text{(no finite age dereliction)}$$

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

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

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

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

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

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

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

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

$$\textsc{Ty-term-FillF}\ \dfrac{\Gamma_1 \vdash t : \lfloor_{n} T\ {}_{m}\!\!\to U \rfloor \qquad \Gamma_2,\ x :_{m} T \vdash u : U}{\Gamma_1 + (1{\uparrow}\cdot n)\cdot\Gamma_2 \vdash t \triangleleft (\lambda x\ {}_{m}\!\!\mapsto u) : 1} \qquad \textsc{Ty-term-FillComp}\ \dfrac{\Gamma_1 \vdash t : \lfloor U \rfloor \qquad \Gamma_2 \vdash t' : U \ltimes T}{\Gamma_1 + 1{\uparrow}\cdot\Gamma_2 \vdash t \triangleleft\!\circ\, t' : T} \qquad \textsc{Ty-term-FillLeaf}\ \dfrac{\Gamma_1 \vdash t : \lfloor_{n} T \rfloor \qquad \Gamma_2 \vdash t' : T}{\Gamma_1 + (1{\uparrow}\cdot n)\cdot\Gamma_2 \vdash t \blacktriangleleft t' : 1}$$

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

$$\textsc{Ty-sterm-FromA'}\ \dfrac{\Gamma \vdash t : T \ltimes 1}{\Gamma \vdash \mathbf{from}'_{\ltimes}\ t : T} \qquad \textsc{Ty-sterm-Unit}\ \dfrac{\texttt{DisposableOnly}\ \Gamma}{\Gamma \vdash () : 1} \qquad \textsc{Ty-sterm-Fun}\ \dfrac{\Gamma_2,\ x :_{m} T \vdash u : U}{\Gamma_2 \vdash \lambda x\ {}_{m}\!\!\mapsto u : T\ {}_{m}\!\!\to U} \qquad \textsc{Ty-sterm-Left}\ \dfrac{\Gamma_2 \vdash t : T_1}{\Gamma_2 \vdash \mathsf{Inl}\,t : T_1 \oplus T_2}$$

$$\textsc{Ty-sterm-Right}\ \dfrac{\Gamma_2 \vdash t : T_2}{\Gamma_2 \vdash \mathsf{Inr}\,t : T_1 \oplus T_2} \qquad \textsc{Ty-sterm-Exp}\ \dfrac{\Gamma_2 \vdash t : T}{m\cdot\Gamma_2 \vdash \mathsf{E}_{m}\ t : !_{m} T} \qquad \textsc{Ty-sterm-Prod}\ \dfrac{\Gamma_{21} \vdash t_1 : T_1 \qquad \Gamma_{22} \vdash t_2 : T_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1\,,\,t_2) : T_1 \otimes T_2}$$

Fig. 6. Typing rules for terms and syntactic sugar

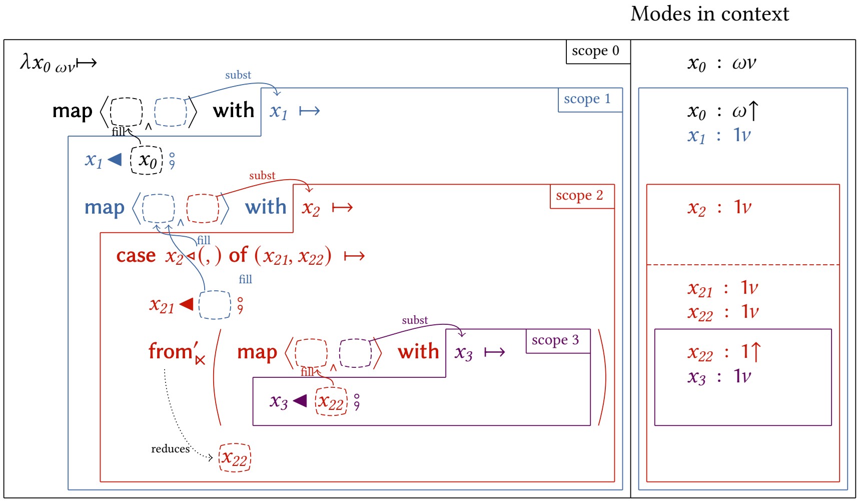

Fig. 7. Scope rules for **map** in $\lambda_d$

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 $\Gamma_1$ 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 $!_{\omega a}(T_1 \otimes T_2) \simeq (!_{\omega a} T_1) \otimes (!_{\omega a} T_2)$, 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, we need a formal notion of *destination scope*. Destination scopes (we'll usually just say *scopes*) are created by rule Ty-term-Map, as destinations are only ever accessed through **map**. More precisely, **map** t **with** $x \mapsto$ t′ creates a new scope for $x$ which spans over t′. In that scope, $x$ has age $\nu$ ("now"), and the age of the other bindings in the context is scaled by $\uparrow$. We see that t′ types in $1\uparrow \cdot \Gamma_2,\ x :_{1\nu} T$ while the global term **map** t **with** $x \mapsto$ t′ mentions unscaled context $\Gamma_2$. 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 filling primitives to avoid pitfalls of Section 3.

Figure 7 illustrates scopes introduced by **map**, and how the typing rules for **map** and ◀ 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, but the structure with holes remains in the outer scope. As a result, when filling a destination with rule Ty-term-FillLeaf, for instance $x_1 ◀ x_0$ in the figure, we type $x_1$ in the new scope, while we type $x_0$ in the outer scope, as it's being moved to the structure with holes on the left of the ampar, which lives in the outer scope too. This is, in fact the opposite of the scaling that **map** does: while **map** creates a new scope for its body, operator ◀, and similarly, ◁∘ and $◁(\lambda x \ _{m}\mapsto$ 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.

When using **from′**⋉ (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

$$
\begin{array}{lcl}
\mathsf{t}, \mathsf{u} & ::= & \ldots \;\mid\; \mathsf{v}
\end{array}
$$

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

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

Fig. 8. Extended terms and runtime values

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 can't contain any scope-controlled resource. This also ensures that the right side cannot contain destinations, so the structure is 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.

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 $\lambda_d$ we need to introduce a few more concepts. We'll need commands $\mathsf{C}\big[\mathsf{t}\big]$, they're described in Section 6.2; and we'll need values, described in Figure 8. Indeed, the terms of $\lambda_d$ 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 $\mathbf{from}'_{\ltimes}$ ($\mathbf{map\ alloc\ with}\ \ldots$)). It's a peculiarity of $\lambda_d$ 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 ${}^{\mathsf{v}}\lambda x\,{}_{\mathsf{m}}\!\mapsto \mathsf{u}$, $\mathsf{u}$ 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

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

$$\textsc{Ty-term-Val}\quad \dfrac{\texttt{DisposableOnly}\ \Gamma \qquad \Delta \;{}^{\mathsf{v}}\!\vdash \mathsf{v} : \mathsf{T}}{\Gamma,\ \Delta \vdash \mathsf{v} : \mathsf{T}}$$

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

$$\textsc{Ty-val-Hole}\quad \dfrac{}{\boxed{h} :_{1\nu} \mathsf{T} \;{}^{\mathsf{v}}\!\vdash \boxed{h} : \mathsf{T}} \qquad \textsc{Ty-val-Dest}\quad \dfrac{1\nu <: \mathsf{m}}{\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \;{}^{\mathsf{v}}\!\vdash \rightarrow\! h : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor}$$

$$\textsc{Ty-val-Unit}\quad \dfrac{}{\bullet \;{}^{\mathsf{v}}\!\vdash () : \mathsf{1}} \qquad \textsc{Ty-val-Fun}\quad \dfrac{\Delta + x :_{\mathsf{m}} \mathsf{T} \vdash \mathsf{u} : \mathsf{U}}{\Delta \;{}^{\mathsf{v}}\!\vdash {}^{\mathsf{v}}\!\lambda x \,{}_{\mathsf{m}}\!\mapsto \mathsf{u} : \mathsf{T} \,{}_{\mathsf{m}}\!\!\to \mathsf{U}}$$

$$\textsc{Ty-val-Left}\quad \dfrac{\Theta \;{}^{\mathsf{v}}\!\vdash \mathsf{v}_1 : \mathsf{T}_1}{\Theta \;{}^{\mathsf{v}}\!\vdash \mathsf{Inl}\,\mathsf{v}_1 : \mathsf{T}_1 \oplus \mathsf{T}_2} \qquad \textsc{Ty-val-Right}\quad \dfrac{\Theta \;{}^{\mathsf{v}}\!\vdash \mathsf{v}_2 : \mathsf{T}_2}{\Theta \;{}^{\mathsf{v}}\!\vdash \mathsf{Inr}\,\mathsf{v}_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}$$

$$\textsc{Ty-val-Prod}\quad \dfrac{\Theta_1 \;{}^{\mathsf{v}}\!\vdash \mathsf{v}_1 : \mathsf{T}_1 \qquad \Theta_2 \;{}^{\mathsf{v}}\!\vdash \mathsf{v}_2 : \mathsf{T}_2}{\Theta_1 + \Theta_2 \;{}^{\mathsf{v}}\!\vdash (\mathsf{v}_1\,,\,\mathsf{v}_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \qquad \textsc{Ty-val-Exp}\quad \dfrac{\Theta \;{}^{\mathsf{v}}\!\vdash \mathsf{v}' : \mathsf{T}}{\mathsf{n}\cdot\Theta \;{}^{\mathsf{v}}\!\vdash \mathsf{E}_{\mathsf{n}}\,\mathsf{v}' : !_{\mathsf{n}} \mathsf{T}}$$

$$\textsc{Ty-val-Ampar}\quad \dfrac{1{\uparrow}\cdot\Delta_1,\ \Delta_3 \;{}^{\mathsf{v}}\!\vdash \mathsf{v}_1 : \mathsf{T} \qquad \Delta_2,\ \rightarrow^{-1}\!\Delta_3 \;{}^{\mathsf{v}}\!\vdash \mathsf{v}_2 : \mathsf{U}}{\Delta_1,\ \Delta_2 \;{}^{\mathsf{v}}\!\vdash {}_{\mathsf{hnames}(\Delta_3)}\langle \mathsf{v}_2 \,{}_{\wedge}\, \mathsf{v}_1 \rangle : \mathsf{U} \ltimes \mathsf{T}}$$

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

of exactly one destination. Thus, the new idea of our system is to feature *hole bindings* $\boxed{h} :_{\mathsf{n}} \mathsf{T}$ and *destination bindings* $\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor$ in typing contexts in addition to the usual variable bindings $x :_{\mathsf{m}} \mathsf{T}$. In both cases, we call $h$ a *hole name*. By definition, a context $\Theta$ 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:

$$\begin{array}{rcl|rcl}
\mathsf{n}'\cdot(\boxed{h} :_{\mathsf{n}} \mathsf{T}) & = & \boxed{h} :_{\mathsf{n}'\cdot\mathsf{n}} \mathsf{T} & (\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\boxed{h} :_{\mathsf{n}'} \mathsf{T}) & = & \boxed{h} :_{\mathsf{n}+\mathsf{n}'} \mathsf{T} \\
\mathsf{n}'\cdot(\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) & = & \rightarrow\! h :_{\mathsf{n}'\cdot\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor & (\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) + (\rightarrow\! h :_{\mathsf{m}'} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) & = & \rightarrow\! h :_{\mathsf{m}+\mathsf{m}'} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \\
 & & & (\boxed{h} :_{\mathsf{n}} \mathsf{T}) + \Theta & = & (\boxed{h} :_{\mathsf{n}} \mathsf{T}),\ \Theta \quad \textit{if}\ h \notin \Theta \\
 & & & (\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) + \Gamma & = & (\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor),\ \Gamma \quad \textit{if}\ h \notin \Gamma
\end{array}$$

Context addition is still very partial; for instance, $(\boxed{h} :_{\mathsf{n}} \mathsf{T}) + (\rightarrow\! h :_{\mathsf{m}} \lfloor_{\mathsf{n}'} \mathsf{T} \rfloor)$ is not defined, as $h$ is present on both sides but with different binding forms.

One of the main goals of $\lambda_d$ 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 $\mathsf{v}_2$ in an ampar ${}_H\langle \mathsf{v}_2 \,{}_{\wedge}\, \mathsf{v}_1 \rangle$, 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's a partial, pointwise operation on typing contexts $\Delta$ defined as: $\rightarrow^{-1}(\rightarrow\! h :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) = \boxed{h} :_{\mathsf{n}} \mathsf{T}$. Note that $\rightarrow^{-1}$ is undefined if any binding has a mode other than $1\nu$.

Furthermore, in rule Ty-val-Ampar, the holes and the corresponding destinations are bound: this is how we ensure that, indeed, there's one destination per hole and one hole per destination. That being said, both sides of the ampar may also contain stored destinations from other scopes, represented by $1{\uparrow}\cdot\Delta_1$ and $\Delta_2$ in the respective typing contexts of $\mathsf{v}_1$ and $\mathsf{v}_2$.

Rule Ty-val-Hole indicates that a hole must have mode $1\nu$ in typing context to be well-typed; in particular weakening and dereliction are not allowed. Only when a hole is behind an exponential, that mode can change to some arbitrary mode $\mathsf{n}$. The mode of a hole restraints which values can be written to it, e.g. in $\boxed{h} :_{\mathsf{n}} \mathsf{T} \;{}^{\mathsf{v}}\!\vdash \mathsf{E}_{\omega\nu}\, \boxed{h} : !_{\mathsf{n}}\mathsf{T}$, only a value with mode $\mathsf{n}$ (more precisely, a value typed in a context of the form $\mathsf{n}\cdot\Theta$) can be written to $\boxed{h}$.

Surprisingly, in Ty-val-Dest, we see that a destination can be typed with any mode $\mathsf{m}$ which $1\nu$ is a subtype of. We did this to mimic the rule Ty-term-Var and make the general modal substitution lemma true for $\lambda_d$[5]. We formally proved however that throughout a well-typed closed program, $\mathsf{m}$ will never be of multiplicity $\omega$ or age $\infty$ — a destination is always linear and of finite age — so mode subtyping is never actually used; and we used this result during the formal proof of the substitution lemma to make it quite easier. The other mode $\mathsf{n}$, appearing in Ty-val-Dest, is not the mode of the destination binding; instead it is part of the type $\lfloor_{\mathsf{n}}\mathsf{T}\rfloor$ and corresponds to the mode of values that we can write to the corresponding $\boxed{h}$; so for it no subtyping can take place.

*Other salient points.* We don't distinguish values with holes from fully-defined values at the syntactic level: instead types prevent holes from being read. In particular, while values are typed in contexts $\Theta$ 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 $\Theta$ 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 ${}^{\mathsf{v}}\lambda x \mapsto \textbf{from}'_{\ltimes}(\textbf{map alloc with } d \mapsto d \triangleleft (\lambda y \mapsto x \textbf{ concat } y))$, where the inner closure is not yet in value form. As the form $d \triangleleft (\lambda y \mapsto x \textbf{ concat } y)$ is part of term syntax, it's allowed to have free variable $x$.

## 6.2 Evaluation contexts and commands

The semantics of $\lambda_d$ is given using small-step reductions on a pair $\mathsf{C}[\mathsf{t}]$ of an evaluation context $\mathsf{C}$, and an (extended) term $\mathsf{t}$ 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 using such commands instead of a store is that destination filling 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 $\mathsf{C}$ is the composition of an arbitrary number of focusing components $\mathsf{c}$. 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 build a stack-based interpreter for the language.

Focusing components are all directly derived from the term syntax, except for the *open ampar* component ${}^{\text{op}}_{H}\langle \mathsf{v}_2 \,{}_{\wedge}\, [\,]\rangle$. This focusing component indicates that an ampar is currently being **map**ped on, with its left-hand side $\mathsf{v}_2$ (the structure being built) being attached to the open ampar focusing component, while its right-hand side (containing destinations) is either in subsequent focusing

[5]Generally, in modal systems, if $x :_{\mathsf{m}} \mathsf{T}$, $\Gamma \vdash \mathsf{u} : \mathsf{U}$ and $\Delta \vdash \mathsf{v} : \mathsf{T}$ then $\mathsf{m}\cdot\Delta, \Gamma \vdash \mathsf{u}[x := \mathsf{v}] : \mathsf{U}$ [Abel and Bernardy 2020]. We have $x :_{\omega\infty} \lfloor_{\mathsf{n}}\mathsf{T}\rfloor \vdash () : 1$ and $\rightarrow h :_{1\nu} \lfloor_{\mathsf{n}}\mathsf{T}\rfloor \vdash \rightarrow h : \lfloor_{\mathsf{n}}\mathsf{T}\rfloor$ so $\omega\infty\cdot(\rightarrow h :_{1\nu} \lfloor_{\mathsf{n}}\mathsf{T}\rfloor) \vdash ()[x := \rightarrow h] : 1$ should be valid.

$$\begin{array}{rcl}
\mathsf{c} & ::= & \mathsf{t}'\;[\,] \;\mid\; [\,]\;\mathsf{v} \;\mid\; [\,]\;\fatsemi\;\mathsf{u} \\
& \mid & \mathbf{case}_{\mathsf{m}}\;[\,]\;\mathbf{of}\;\{\mathsf{Inl}\;x_1 \mapsto \mathsf{u}_1\,,\;\mathsf{Inr}\;x_2 \mapsto \mathsf{u}_2\} \;\mid\; \mathbf{case}_{\mathsf{m}}\;[\,]\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto \mathsf{u} \;\mid\; \mathbf{case}_{\mathsf{m}}\;[\,]\;\mathbf{of}\;\mathsf{E}_{\mathsf{n}}\;x \mapsto \mathsf{u} \\
& \mid & \mathbf{map}\;[\,]\;\mathbf{with}\;x \mapsto \mathsf{t}' \;\mid\; \mathbf{to}_{\ltimes}\;[\,] \;\mid\; \mathbf{from}_{\ltimes}\;[\,] \;\mid\; [\,] \triangleleft\!\circ\, \mathsf{t}' \;\mid\; \mathsf{v} \triangleleft\!\circ [\,] \;\mid\; [\,] \blacktriangleleft \mathsf{t}' \;\mid\; \mathsf{v} \blacktriangleleft [\,] \\
& \mid & [\,] \triangleleft () \;\mid\; [\,] \triangleleft \mathsf{Inl} \;\mid\; [\,] \triangleleft \mathsf{Inr} \;\mid\; [\,] \triangleleft (,) \;\mid\; [\,] \triangleleft \mathsf{E}_{\mathsf{m}} \;\mid\; [\,] \triangleleft (\lambda x \;{}_{\mathsf{m}}\!\mapsto \mathsf{u}) \\
& \mid & {}^{\mathrm{op}}_{H}\langle \mathsf{v}_2 \,{}_{\wedge}\, [\,] \rangle \qquad \textit{(open ampar focusing component)} \\[1em]
\mathsf{C} & ::= & [\,] \;\mid\; \mathsf{C} \;\circ\; \mathsf{c} \;\mid\; \mathsf{C}\,(\!| h :=_{H} \;\mathsf{v} |\!)
\end{array}$$

Fig. 10. Grammar for evaluation contexts

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 $\mathsf{C}\,(\!| h :=_{H} \;\mathsf{v} |\!)$ that is used to update structures under construction that are attached to open ampar focusing components in the stack. Such a substitution is triggered when a destination $\rightarrow\!h$ is filled in the term under focus, and results in the value $\mathsf{v}$ (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 $\mathsf{v}$, and is used to update the hole name set of the ampar:

$$\begin{array}{rcll}
(\mathsf{C} \;\circ\; {}^{\mathrm{op}}_{\{h\}\sqcup H}\langle \mathsf{v}_2 \,{}_{\wedge}\, [\,]\rangle)\,(\!| h :=_{H'} \;\mathsf{v}' |\!) & = & \mathsf{C} \;\circ\; {}^{\mathrm{op}}_{H \sqcup H'}\langle \mathsf{v}_2\,(\!| h :=_{H'} \;\mathsf{v}' |\!) \,{}_{\wedge}\, [\,]\rangle & \\
(\mathsf{C} \;\circ\; \mathsf{c})\,(\!| h :=_{H'} \;\mathsf{v}' |\!) & = & \mathsf{C}\,(\!| h :=_{H'} \;\mathsf{v}' |\!) \;\circ\; \mathsf{c} & \textit{otherwise}
\end{array}$$

Evaluation contexts $\mathsf{C}$ are typed in a context $\Delta$ that can only contains destination bindings. As we can see in rule Ty-cmd, $\Delta$ is exactly the typing context that the term $\mathsf{t}$ has to use to form the command $\mathsf{C}\big[\mathsf{t}\big]$. In other words, while $\Gamma \vdash \mathsf{t} : \mathsf{T}$ *requires* the bindings of $\Gamma$, judgment $\Delta \dashv \mathsf{C} : \mathsf{T} \rightarrowtail \mathsf{U}_0$ *provides* the bindings of $\Delta$. Typing rules for evaluation contexts and commands are given in Figure 11.

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

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

$$\textsc{Ty-ectxs-PatP}\;\frac{x_1 :_{\mathsf{m}} \mathsf{T}_1 \;\#\; x_2 :_{\mathsf{m}} \mathsf{T}_2 \qquad \mathsf{m}\!\cdot\!\Delta_1,\; \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2 + x_1 :_{\mathsf{m}} \mathsf{T}_1 + x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u} : \mathsf{U}}{\Delta_1 \dashv \mathsf{C} \;\circ\; \mathbf{case}_{\mathsf{m}}\;[\,]\;\mathbf{of}\;(x_1\,,\,x_2) \mapsto \mathsf{u} : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

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

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

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

$$\textsc{Ty-ectxs-Id}\quad \frac{}{\cdot \dashv [\,] : \mathsf{U}_0 \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-App1}\quad \frac{\begin{array}{c} \mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_2 \vdash \mathsf{t}' : \mathsf{T}\ {}_{\mathsf{m}}\!\!\to \mathsf{U} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ \mathsf{t}'\ [\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-App2}\quad \frac{\begin{array}{c} \mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_1 \vdash \mathsf{v} : \mathsf{T} \end{array}}{\Delta_2 \dashv \mathsf{C} \circ [\,]\ \mathsf{v} : (\mathsf{T}\ {}_{\mathsf{m}}\!\!\to \mathsf{U}) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatU}\quad \frac{\begin{array}{c} \Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_2 \vdash \mathsf{u} : \mathsf{U} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ [\,] \mathbin{;} \mathsf{u} : 1 \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatS}\quad \frac{\begin{array}{c} \mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_2 + x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash \mathsf{u}_1 : \mathsf{U} \qquad \Delta_2 + x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u}_2 : \mathsf{U} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ \mathbf{case}_{\mathsf{m}}\ [\,]\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto \mathsf{u}_1\, ,\ \mathsf{Inr}\, x_2 \mapsto \mathsf{u}_2\} : (\mathsf{T}_1 \oplus \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatP}\quad \frac{\begin{array}{c} x_1 :_{\mathsf{m}} \mathsf{T}_1\ \#\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \\ \mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_2 + x_1 :_{\mathsf{m}} \mathsf{T}_1 + x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash \mathsf{u} : \mathsf{U} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ \mathbf{case}_{\mathsf{m}}\ [\,]\ \mathbf{of}\ (x_1\, ,\, x_2) \mapsto \mathsf{u} : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-PatE}\quad \frac{\begin{array}{c} \mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \rightarrowtail \mathsf{U}_0 \\ \Delta_2 + x :_{\mathsf{m}\cdot\mathsf{m}'} \mathsf{T} \vdash \mathsf{u} : \mathsf{U} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ \mathbf{case}_{\mathsf{m}}\ [\,]\ \mathbf{of}\ \mathrm{E}_{\mathsf{m}'}\ x \mapsto \mathsf{u} : !_{\mathsf{m}'} \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-Map}\quad \frac{\begin{array}{c} \Delta_1,\ \Delta_2 \dashv \mathsf{C} : \mathsf{U} \ltimes \mathsf{T}' \rightarrowtail \mathsf{U}_0 \\ 1\!\uparrow\cdot\Delta_2 + x :_{1\nu} \mathsf{T} \vdash \mathsf{t}' : \mathsf{T}' \end{array}}{\Delta_1 \dashv \mathsf{C} \circ \mathbf{map}\ [\,]\ \mathbf{with}\ x \mapsto \mathsf{t}' : (\mathsf{U} \ltimes \mathsf{T}) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-ToA}\quad \frac{\Delta \dashv \mathsf{C} : (\mathsf{U} \ltimes 1) \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ \mathbf{to}_{\ltimes}\ [\,] : \mathsf{U} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FromA}\quad \frac{\Delta \dashv \mathsf{C} : (\mathsf{U} \otimes (!_{1\infty} \mathsf{T})) \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ \mathbf{from}_{\ltimes}\ [\,] : (\mathsf{U} \ltimes (!_{1\infty} \mathsf{T})) \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillU}\quad \frac{\Delta \dashv \mathsf{C} : 1 \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ [\,] \triangleleft () : \lfloor_{\mathsf{n}} 1 \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillL}\quad \frac{\Delta \dashv \mathsf{C} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ [\,] \triangleleft \mathsf{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillR}\quad \frac{\Delta \dashv \mathsf{C} : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ [\,] \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillP}\quad \frac{\Delta \dashv \mathsf{C} : (\lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor) \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ [\,] \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillE}\quad \frac{\Delta \dashv \mathsf{C} : \lfloor_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv \mathsf{C} \circ [\,] \triangleleft \mathrm{E}_{\mathsf{m}} : \lfloor_{\mathsf{n}} !_{\mathsf{m}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillF}\quad \frac{\begin{array}{c} \Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv \mathsf{C} : 1 \rightarrowtail \mathsf{U}_0 \\ \Delta_2 + x :_{\mathsf{m}} \mathsf{T} \vdash \mathsf{u} : \mathsf{U} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ [\,] \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\!\mapsto \mathsf{u}) : \lfloor_{\mathsf{n}} \mathsf{T}\ {}_{\mathsf{m}}\!\!\to \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillComp1}\quad \frac{\begin{array}{c} \Delta_1,\ 1\!\uparrow\cdot\Delta_2 \dashv \mathsf{C} : \mathsf{T} \rightarrowtail \mathsf{U}_0 \\ \Delta_2 \vdash \mathsf{t}' : \mathsf{U} \ltimes \mathsf{T} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ [\,] \triangleleft\!\circ\, \mathsf{t}' : \lfloor \mathsf{U} \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillComp2}\quad \frac{\begin{array}{c} \Delta_1,\ 1\!\uparrow\cdot\Delta_2 \dashv \mathsf{C} : \mathsf{T} \rightarrowtail \mathsf{U}_0 \\ \Delta_1 \vdash \mathsf{v} : \lfloor \mathsf{U} \rfloor \end{array}}{\Delta_2 \dashv \mathsf{C} \circ \mathsf{v} \triangleleft\!\circ [\,] : \mathsf{U} \ltimes \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillLeaf1}\quad \frac{\begin{array}{c} \Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv \mathsf{C} : 1 \rightarrowtail \mathsf{U}_0 \\ \Delta_2 \vdash \mathsf{t}' : \mathsf{T} \end{array}}{\Delta_1 \dashv \mathsf{C} \circ [\,] \blacktriangleleft \mathsf{t}' : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-FillLeaf2}\quad \frac{\begin{array}{c} \Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv \mathsf{C} : 1 \rightarrowtail \mathsf{U}_0 \\ \Delta_1 \vdash \mathsf{v} : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \end{array}}{\Delta_2 \dashv \mathsf{C} \circ \mathsf{v} \blacktriangleleft [\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}$$

$$\textsc{Ty-ectxs-OpenAmpar}\quad \frac{\begin{array}{c} \mathsf{hnames}(\mathsf{C})\ \#\#\ \mathsf{hnames}(\Delta_3) \\ \Delta_1,\ \Delta_2 \dashv \mathsf{C} : (\mathsf{U} \ltimes \mathsf{T}') \rightarrowtail \mathsf{U}_0 \qquad \Delta_2,\ \rightarrow^{-1}\Delta_3 \,{}^{\mathsf{v}}\!\!\vdash \mathsf{v}_2 : \mathsf{U} \end{array}}{1\!\uparrow\cdot\Delta_1,\ \Delta_3 \dashv \mathsf{C} \circ {}^{\mathrm{op}}_{\mathsf{hnames}(\Delta_3)}\langle \mathsf{v}_2 \,{}_{\wedge}\, [\,] \rangle : \mathsf{T}' \rightarrowtail \mathsf{U}_0}$$

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

$$\textsc{Ty-cmd}\quad \frac{\Delta \dashv \mathsf{C} : \mathsf{T} \rightarrowtail \mathsf{U}_0 \qquad \Delta \vdash \mathsf{t} : \mathsf{T}}{\vdash \mathsf{C}[\mathsf{t}] : \mathsf{U}_0}$$

Fig. 11. Typing rules for evaluation contexts and commands

## 6.3 Small-step semantics

We equip $\lambda_d$ 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 $\mathrm{C}$;
- unfocus rules, where the term under focus is in fact a value, and thus we pop a focusing component from the stack $\mathrm{C}$ 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:

$$\begin{array}{lll}
\mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ (x_1\,,x_2) \mapsto \mathrm{u}\big] & \longrightarrow & \big(\mathrm{C}\ \circ\ \mathbf{case}_{\mathrm{m}}\ [\,]\ \mathbf{of}\ (x_1\,,x_2) \mapsto \mathrm{u}\big)\big[\mathrm{t}\big] \quad \textit{when} \quad \mathtt{NotVal}\ \mathrm{t} \\
\big(\mathrm{C}\ \circ\ \mathbf{case}_{\mathrm{m}}\ [\,]\ \mathbf{of}\ (x_1\,,x_2) \mapsto \mathrm{u}\big)\big[\mathrm{v}\big] & \longrightarrow & \mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ \mathrm{v}\ \mathbf{of}\ (x_1\,,x_2) \mapsto \mathrm{u}\big] \\
\mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ (\mathrm{v}_1\,,\mathrm{v}_2)\ \mathbf{of}\ (x_1\,,x_2) \mapsto \mathrm{u}\big] & \longrightarrow & \mathrm{C}\big[\mathrm{u}[x_1 \coloneqq \mathrm{v}_1][x_2 \coloneqq \mathrm{v}_2]\big]
\end{array}$$

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 (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 $\boxed{h}$; we model this as a special substitution $\mathrm{C}\,(\!|h \coloneqq_H\ \mathrm{v}|\!)$ on the evaluation context $\mathrm{C}$. 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 $\mathrm{C}\,(\!|h \coloneqq_H\ \mathrm{v}|\!)$ should only be performed if $h$ is a globally unique name; otherwise we break the promise of a write-once memory model. To this effect, we allow name shadowing while an ampar is closed, but as soon as an ampar is open, it should have globally unique hole names. This restriction is enforced by premise $\mathsf{hnames}(\mathrm{C})\ \#\#\ \mathsf{hnames}(\Delta_3)$ 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 $h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}) \cup \{h\}) + 1$ or similar definitions for $h''$ and $h'''$ (see in Section 6.3) to find the next unused name. Then we use either the *shifted set* $H \pm h' \triangleq \{h + h' \mid h \in H\}$ or the *conditional shift operator*:

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

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

Rules Open-Ampar and Close-Ampar dictate how and when a closed ampar (a value) is converted to an open ampar (a focusing component) and vice-versa, and they make use of the shifting strategy we've just introduced. With Open-Ampar, the hole names bound by the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing component ${}^{\mathrm{op}}_{H \pm h'}\langle \mathrm{v}_2[H \pm h''']\ {}_{\wedge}\, [\,]\rangle$ 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

$$\boxed{\mathrm{C}[\mathrm{t}] \longrightarrow \mathrm{C}'[\mathrm{t}']} \qquad \textit{(Small-step evaluation of commands)}$$

$$\begin{array}{ll}
\mathrm{C}\big[({}^{\mathsf{v}}\lambda x\,{}_{\mathrm{m}}\!\mapsto \mathrm{u})\ \mathrm{v}\big] \longrightarrow \mathrm{C}\big[\mathrm{u}[x \coloneqq \mathrm{v}]\big] & \textsc{Red-App} \\
\mathrm{C}\big[()\ ;\ \mathrm{u}\big] \longrightarrow \mathrm{C}\big[\mathrm{u}\big] & \textsc{Red-PatU} \\
\mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ (\mathsf{Inl}\,\mathrm{v}_1)\ \mathbf{of}\ \{\mathsf{Inl}\,x_1 \mapsto \mathrm{u}_1,\ \mathsf{Inr}\,x_2 \mapsto \mathrm{u}_2\}\big] \longrightarrow \mathrm{C}\big[\mathrm{u}_1[x_1 \coloneqq \mathrm{v}_1]\big] & \textsc{Red-PatL} \\
\mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ (\mathsf{Inr}\,\mathrm{v}_2)\ \mathbf{of}\ \{\mathsf{Inl}\,x_1 \mapsto \mathrm{u}_1,\ \mathsf{Inr}\,x_2 \mapsto \mathrm{u}_2\}\big] \longrightarrow \mathrm{C}\big[\mathrm{u}_2[x_2 \coloneqq \mathrm{v}_2]\big] & \textsc{Red-PatR} \\
\mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ (\mathrm{v}_1, \mathrm{v}_2)\ \mathbf{of}\ (x_1, x_2) \mapsto \mathrm{u}\big] \longrightarrow \mathrm{C}\big[\mathrm{u}[x_1 \coloneqq \mathrm{v}_1][x_2 \coloneqq \mathrm{v}_2]\big] & \textsc{Red-PatP} \\
\mathrm{C}\big[\mathbf{case}_{\mathrm{m}}\ \mathsf{E}_{\mathrm{n}}\,\mathrm{v}'\ \mathbf{of}\ \mathsf{E}_{\mathrm{n}}\,x \mapsto \mathrm{u}\big] \longrightarrow \mathrm{C}\big[\mathrm{u}[x \coloneqq \mathrm{v}']\big] & \textsc{Red-PatE} \\
\mathrm{C}\big[\mathbf{to}_{\ltimes}\ \mathrm{v}_2\big] \longrightarrow \mathrm{C}\big[{}_{\{\,\}}\langle \mathrm{v}_2 \,{}_{\wedge}\, () \rangle\big] & \textsc{Red-ToA} \\
\mathrm{C}\big[\mathbf{from}_{\ltimes}\ {}_{\{\,\}}\langle \mathrm{v}_2 \,{}_{\wedge}\, \mathsf{E}_{1\infty}\,\mathrm{v}_1 \rangle\big] \longrightarrow \mathrm{C}\big[(\mathrm{v}_2, \mathsf{E}_{1\infty}\,\mathrm{v}_1)\big] & \textsc{Red-FromA} \\
\mathrm{C}\big[\mathbf{alloc}\big] \longrightarrow \mathrm{C}\big[{}_{\{1\}}\langle \boxed{1} \,{}_{\wedge} \rightarrow 1 \rangle\big] & \textsc{Red-Alloc} \\
\mathrm{C}\big[\rightarrow h \triangleleft ()\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{\,\}}\ ()\,|\!)\big[()\big] & \textsc{Red-FillU} \\
\mathrm{C}\big[\rightarrow h \triangleleft (\lambda x\,{}_{\mathrm{m}}\!\mapsto \mathrm{u})\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{\,\}}\ {}^{\mathsf{v}}\lambda x\,{}_{\mathrm{m}}\!\mapsto \mathrm{u}\,|\!)\big[()\big] & \textsc{Red-FillF} \\
\mathrm{C}\big[\rightarrow h \triangleleft \mathsf{Inl}\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{h'+1\}}\ \mathsf{Inl}\,\boxed{h'+1}\,|\!)\big[\rightarrow h'+1\big] & \textsc{Red-FillL} \\
\mathrm{C}\big[\rightarrow h \triangleleft \mathsf{Inr}\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{h'+1\}}\ \mathsf{Inr}\,\boxed{h'+1}\,|\!)\big[\rightarrow h'+1\big] & \textsc{Red-FillR} \\
\mathrm{C}\big[\rightarrow h \triangleleft \mathsf{E}_{\mathrm{m}}\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{h'+1\}}\ \mathsf{E}_{\mathrm{m}}\,\boxed{h'+1}\,|\!)\big[\rightarrow h'+1\big] & \textsc{Red-FillE} \\
\mathrm{C}\big[\rightarrow h \triangleleft (,)\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{h'+1, h'+2\}}\ (\boxed{h'+1}, \boxed{h'+2})\,|\!)\big[(\rightarrow h'+1, \rightarrow h'+2)\big] & \textsc{Red-FillP} \\
\mathrm{C}\big[\rightarrow h \triangleleft\!\circ_{H} \langle \mathrm{v}_2 \,{}_{\wedge}\, \mathrm{v}_1 \rangle\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{(H \pm h'')}\ \mathrm{v}_2[H \pm h'']\,|\!)\big[\mathrm{v}_1[H \pm h'']\big] & \textsc{Red-FillComp} \\
\mathrm{C}\big[\rightarrow h \blacktriangleleft \mathrm{v}\big] \longrightarrow \mathrm{C}\,(\!|h \coloneqq_{\{\,\}}\ \mathrm{v}\,|\!)\big[()\big] & \textsc{Red-FillLeaf} \\
\mathrm{C}\big[\mathbf{map}\ {}_{H}\langle \mathrm{v}_2 \,{}_{\wedge}\, \mathrm{v}_1 \rangle\ \mathbf{with}\ x \mapsto \mathrm{t}'\big] \longrightarrow \big(\mathrm{C} \circ {}^{\mathrm{op}}_{H \pm h'''}\langle \mathrm{v}_2[H \pm h'''] \,{}_{\wedge}\, [] \rangle\big)\big[\mathrm{t}'[x \coloneqq \mathrm{v}_1[H \pm h''']]\big] & \textsc{Open-Ampar} \\
\big(\mathrm{C} \circ {}^{\mathrm{op}}_{H}\langle \mathrm{v}_2 \,{}_{\wedge}\, [] \rangle\big)\big[\mathrm{v}_1\big] \longrightarrow \mathrm{C}\big[{}_{H}\langle \mathrm{v}_2 \,{}_{\wedge}\, \mathrm{v}_1 \rangle\big] & \textsc{Close-Ampar}
\end{array}$$

$$\textit{where} \quad \left\{ \begin{array}{lcl}
h' & = & \max(\mathbf{hnames}(\mathrm{C}) \cup \{h\}) + 1 \\
h'' & = & \max(H \cup (\mathbf{hnames}(\mathrm{C}) \cup \{h\})) + 1 \\
h''' & = & \max(H \cup \mathbf{hnames}(\mathrm{C})) + 1
\end{array} \right.$$

Fig. 12. Small-step semantics

**map** statement has reduced to a value. In that case, we can close the ampar, by popping the focusing component from the stack $\mathrm{C}$ and merging back with $\mathrm{v}_2$ to form a closed ampar again.

In rule Red-FillComp, we write the left-hand side $\mathrm{v}_2$ of a closed ampar ${}_{H}\langle \mathrm{v}_2 \,{}_{\wedge}\, \mathrm{v}_1 \rangle$ to a hole $\boxed{h}$ that is part of a structure with holes somewhere inside $\mathrm{C}$. This results in the composition of two structures with holes. Because we dissociate $\mathrm{v}_2$ and $\mathrm{v}_1$ that were previously bound together by the ampar connective ($\mathrm{v}_2$ is merged with another structure, while $\mathrm{v}_1$ becomes the new focus), their hole names are no longer bound, so we need to make them globally unique, as we do when an ampar is opened with **map**. This renaming is carried out by the conditional shift $\mathrm{v}_2[H \pm h'']$ and $\mathrm{v}_1[H \pm h'']$.

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

Theorem 6.1 (Type preservation). *If* $\vdash \mathrm{C}[\mathrm{t}] : \mathrm{T}$ *and* $\mathrm{C}[\mathrm{t}] \longrightarrow \mathrm{C}'[\mathrm{t}']$ *then* $\vdash \mathrm{C}'[\mathrm{t}'] : \mathrm{T}$.

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

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

## 7 FORMAL PROOF OF TYPE SAFETY

We've proved type preservation and progress theorems with the Coq proof assistant. Turning to a proof assistant was a pragmatic choice: typing context handling in $\lambda_d$ 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 $\lambda_d$ 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 7000 lines long, and contains nearly 500 lemmas. Many of the cases of the type preservation and progress lemmas are similar. To handle such repetitive cases, the use of a large-language-model based autocompletion system has proven quite effective.

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 }.
```

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 $\lambda_d$ USING IN-PLACE MEMORY MUTATIONS

The formal language presented in Sections 5 and 6 is not meant to be implemented as-is.

First, $\lambda_d$ 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 $\lambda_d$; only destinations are. That is to say that an ampar can be wrapped in an exponential, e.g. $\mathrm{E}_{\omega\nu}\ {}_{\{h\}}\langle 0 :: \boxed{h}\ {}_{\wedge}\!\rightarrow\! h\rangle$ (representing a non-linear difference list $0{::}\square$), and then used twice, each time in a different way:

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

It may seem counter-intuitive at first, but this program is valid and safe in $\lambda_d$. 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 [Lorenzen et al. 2023], where, thanks to 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. We held back from doing that in formalization of $\lambda_d$ as it obfuscates the presentation of the system without adding much to the understanding of the latter.

To make ampars linear, we follow a recipe proposed by [Spiwack et al. 2022] and introduce a new type Token, together with primitives **dup** and **drop**. We also switch **alloc** for $\textbf{alloc}_{\textbf{ip}}$:

$$\begin{array}{lll}\textbf{dup} & : & \mathsf{Token} \rightarrow \mathsf{Token} \otimes \mathsf{Token} \qquad \text{(remember that unqualified arrows} \rightarrow \text{have mode } 1\nu\text{, so are linear)} \\ \textbf{drop} & : & \mathsf{Token} \rightarrow 1 \\ \textbf{alloc}_{\textbf{ip}} & : & \mathsf{Token} \rightarrow \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor\end{array}$$

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

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

An alternative to having a linear root token variable is to add a primitive function
$\textbf{withToken}\ :\ (\mathsf{Token}\ {}_{1\infty}\!\!\rightarrow\ !_{\omega\infty}\mathsf{T}) \rightarrow\ !_{\omega\infty}\mathsf{T}$ that fulfill the same goal.

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

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

$\lambda_d$ extended with Tokens and $\textbf{alloc}_{\textbf{ip}}$ is in fact very close to the implementation described in [Bagrel 2024]. Our claim of efficiency is thus based on the results published in the latter and also [Bour et al. 2021; Lorenzen et al. 2024], as an hypothetical implementation of $\lambda_d$ would mostly resort to the same memory operations – that is, in-place updates in recursive functional settings.

## 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 languages, although their work is on array programming, while this article focuses on linked data structures. 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 see these two aspects as complementing 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], it is entirely predictable.

This has been available in OCaml since version 4.14. This is the one example we know of of destinations built in a production-grade compiler. Our $\lambda_d$ 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 $\mathsf{S} \ltimes \mathsf{T}$, 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 $\lambda_d$, 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 elements give extra expressiveness to the language compared to [Minamide 1998].

In their system, $d \blacktriangleleft \mathsf{t}$ requires $\mathsf{t}$ to be unrestricted, while in $\lambda_d$, $\mathsf{t}$ can be linear. The consequence is that in [Bagrel 2024], destinations can be stored in data structures but not in data structures with holes; so in a breadth-first search algorithm like in Section 4, they have to build the queue using normal constructors, and cannot use destination-filling primitives. Therefore both normal constructors and DPS primitives must coexist in their work, while in $\lambda_d$, only DPS primitives are required to bootstrap the system, as we later derive normal constructors from them.

However, [Bagrel 2024] only requires a linear type system such as the one of Haskell to work. 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 semantics of holes is that of a future or a promise. In turns this requires the semantics of their calculus to be fully concurrent. Which is a very different point in the design space.

## 10 CONCLUSION AND FUTURE WORK

Using a system of ages in addition to linearity, $\lambda_d$ 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 $\lambda_d$'s flexibility.

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

## DATA-AVAILABILITY STATEMENT

In Section 7, we mentioned that we've proved type preservation and progress theorems with the Coq proof assistant. The Coq code is available at https://doi.org/10.5281/zenodo.13933661, bundled with a Nix flake file to set up the environment to run the proof.

## 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. *arXiv:2102.09823 [cs]* (Feb. 2021). http://arxiv.org/abs/2102.09823 arXiv: 2102.09823.

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