Destination calculus

A linear λ−calculus for pure, functional memory updates

ARNAUD SPIWACK, Tweag, France
THOMAS BAGREL, LORIA/Inria, France and Tweag, France

We present the λ_d , a linear λ−calculus for pure, functional memory updates. We introduce the syntax, type system, and operational semantics of the destination calculus, and prove type safety formally in the Coq proof assistant.

We show how the principles of the λ_d can form a theoretical ground for destination-passing style programming in functional languages. In particular, we detail how the present work can be applied to Linear Haskell to lift the main restriction of DPS programming in Haskell as developed in [Bagrel 2024]. We illustrate this with a range of pseudo-Haskell examples.

ACM Reference Format:
Arnaud Spiwack and Thomas Bagrel. 2024. Destination calculus: A linear λ−calculus for pure, functional memory updates. In . ACM, New York, NY, USA, 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

TODO: Redire plein de fois: On introduit de la mutation controlee dans les FP languages sans endommager la purete (comme la lazyness peut être vu aussi) + ordre de building flexible + des systemes se sont deja interesses a ca, mais nous on subsume tout ca

Destination-passing style programming takes its root in the early days of imperative programming. In such language, the programmer is responsible for managing memory allocation and deallocation, and thus is it often unpractical for function calls to allocate memory for their results themselves. Instead, the caller allocates memory for the result of the callee, and passes the address of this output memory cell to the callee as an argument. This is called an *out parameter*, *mutable reference*, or even *destination*.

But destination-passing style is not limited to imperative settings; it can be used in functional programming as well. One example is the linear destination-based API for arrays in Haskell[Bernardy et al. 2018], which enables the user to build an array efficiently in a write-once fashion, without sacrificing the language identity and main guarantees. In this context, a destination points to a yet-unfilled memory slot of the array, and is said to be *consumed* as soon as the associated hole is written to. In this paper, we continue on the same line: we present a linear λ−calculus embedding the concept of *destinations* as first-class values, in order to provide a write-once memory scheme for pure, functional programming languages.

Why is it important to have destinations as first-class values? Because it allows the user to store them in arbitrary control or data structures, and thus to build complex data structures in arbitrary order/direction. This is a key feature of first-class DPS APIs, compared to ones in which destinations are inseparable from the structure they point to. In the latter case, the user is still forced to build

POPL'25, January 19 – 25, 2025, Denver, Colorado

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the structure in its canonical order (e.g. from the leaves up to the root of the structure when using data constructors).

2 WORKING WITH DESTINATIONS

TODO: Some introductory words

2.1 Building up a vocabulary

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

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

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

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

Maybe it would be wise to explain the notations for sums and tuples, since the linear-logic notations are less standard than the ccc ones

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

In this example, we're building a value of type $\mathsf{T} \oplus \mathsf{U}$ by setting the outermost constructor to Inl. We think of $d \triangleleft \mathsf{Inl}$ as allocating memory to store a block of the form Inl $\square$, write the address of that block to the location that d points to, and return a destination pointing to the uninitialized argument of Inl.

We probably want to give a reading for the fill-with-a-constructor construction.

Notice that we are constructing the structure from the outermost constructor inward: we've built a value of the form Inl $\square$, but we have yet to describe what goes in the hole $\square$. We call such incomplete values *hollow constructors*. This is opposite to how functional programming usually works, where values are built from the innermost constructors outward: first we make a value v and only then can we use Inl to make an Inl v. This will turn out to be a key ingredient in the expressiveness of destination passing.

Yet, everything we've shown so far could have been done with continuations. So it's worth asking: how are destination different from continuations? Part of the answer lies in our intention to represent destinations as pointers to uninitialized memory (see Section 8). But where destinations really differ from continuations is when there are several destinations. Then you can (indeed you must!) fill all the destinations; whereas when you have multiple continuations, you can only return to one of them. Multiple destination arises from filling destination of tuples:

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

To fill a destination for a pair, we must fill both the first field and the second field. In plain English, it sounds obvious, but the key remark is that **fillWithAPair** doesn't exist on continuations.

Structures with holes. Let's now turn to how we can use the result made by filling destinations. Observe, as a preliminary remark, 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. For instance, **fillWithInl**, above, returns a destination $\lfloor \mathsf{T} \rfloor$ while it is used to build a structure of type $\mathsf{T} \oplus \mathsf{U}$. To represents this, λ_d uses a type $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ for a structure of type S missing a value of type T to be complete; we

Make sure that we clarify early on that when we say "structure" we mean "linked data structure" then say it has a *hole* of type T. There can be several holes in S, in which case the right-hand side is a tuple of destinations: $\mathsf{S} \ltimes (\lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor)$.

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

Probably not the right label to point to. Make sure that we clarify the differences with a par.

Destinations always exist within the context of a structure with holes: a destination is a pointer to a hole in the structure. Crucially, destinations are otherwise ordinary values. To access the destinations, λ_d provides a **map** construction, which lets us apply a function to the right-hand side of an ampar:

$$\begin{array}{ll}
\mathsf{fillWithInl'} \;:\; \mathsf{S} \ltimes \lfloor \mathsf{T} \oplus \mathsf{U} \rfloor \to \mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor & \mathsf{fillWithAPair'} \;:\; \mathsf{S} \ltimes \lfloor \mathsf{T} \otimes \mathsf{U} \rfloor \to \mathsf{S} \ltimes (\lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor) \\
\mathsf{fillWithInl'}\; x \;\triangleq\; \textbf{map}\; x \;\textbf{with}\; d \mapsto d \triangleleft \mathsf{Inl} & \mathsf{fillWithAPair'}\; x \;\triangleq\; \textbf{map}\; x \;\textbf{with}\; d \mapsto d \triangleleft (,)
\end{array}$$

To tie this up, we need a way to introduce and to eliminate structures with holes. Structures with holes are introduced with alloc which creates a value of type $\mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor$. alloc is a bit like the identity function: it is a hole of type T that needs a structure of type T to be a complete structure of type T. Structures with holes are eliminated with[1] $\textbf{from}'_{\ltimes} : \mathsf{S} \ltimes 1 \to \mathsf{S}$: if all the destinations have been consumed and only unit remains on the right side, then a structure with holes is really just a normal, complete structure.

Equipped with these, we can, for instance, derive traditional constructors from piecemeal filling. In fact, λ_d doesn't have primitive constructor forms, constructors in λ_d are syntactic sugar. We show here the definition of Inl and (,), but the other constructors are derived similarly.

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

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

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

Ensure that we do claim that

- Destination are written at least once, preventing examples like

  $$\begin{array}{l}
  \mathsf{forget} \;:\; \mathsf{T} \\
  \mathsf{forget} \;\triangleq\; \textbf{from}'_{\ltimes}\,(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto ())
  \end{array}$$

  where reading the result of **forget** would result in reading a hole that we never filled, in other words, reading uninitialized memory.
- Destination are written at most once, preventing examples like

  $$\begin{array}{l}
  \mathsf{ambiguous1} \;:\; \mathsf{Bool} \\
  \mathsf{ambiguous1} \;\triangleq\; \textbf{from}'_{\ltimes}\,(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto d \blacktriangleleft \mathsf{true} \,⨟\, d \blacktriangleleft \mathsf{false})
  \end{array}$$

  $$\begin{array}{l}
  \mathsf{ambiguous2} \;:\; \mathsf{Bool} \\
  \mathsf{ambiguous2} \;\triangleq\; \textbf{from}'_{\ltimes}\,(\textbf{map}\; \mathsf{alloc}\; \textbf{with}\; d \mapsto \textbf{let}\; x := (d \blacktriangleleft \mathsf{false})\; \textbf{in}\; d \blacktriangleleft \mathsf{true} \,⨟\, x)
  \end{array}$$

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

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

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

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

Fig. 1. List implementation in equirecursive λ_d

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

2.2 Functional queues, with destinations

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

Linked lists. For starters, we can define lists as the fixpoint of the functor $X \mapsto 1 \oplus (\mathsf{T} \otimes X)$ where T is the type of list items. Following the recipe that we've outline so far, instead of defining the usual "nil" [] and "cons" (::) constructors, we define the more general "fillNil" $\triangleleft[\,]$ and "fillCons" $\triangleleft(::)$ operators, as presented in Figure 1.

Arnaud hasn't used in the previous section small-caps name for constructors and operations. If we are to use them in the next few sections, it would be good to introduce this vocabulary early.

Just like we did in Section 2.1 for the primitive constructors, we can recover the "cons" constructor:

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

Going from a "fill" operator to the associated constructor is completely generic, and with more metaprogramming tools, we could build this transformation into the language.

Difference lists. While linked lists are optimized for the prepend operation ("cons"), they are not efficient for appending or concatenation, as it requires a full copy (or traversal at least) of the first list before the last cons cell can be changed to point to the head of the second list.

Difference lists are a data structure that allows for efficient concatenation. In functional languages, difference lists are often encoded using a function that take a tail, and returns the previously-unfinished list with the tail appended to it. For example, the difference list $x_1 {:} x_2 {:} \ldots {:} x_k {:} \square$ is represented by the linear function $\lambda xs \mapsto x_1 {:} x_2 {:} \ldots {:} x_k {:} xs$. This encoding shines when list concatenation calls are nested to the left, as the function encoding delays the actual concatenation so that it happens in a more optimal, right-nested fashion than a naive nested list concatenation.

In λ_d , we can go even further, and represent difference lists much like we would do in an imperative programming language (although in a safe setting here), as a pair of an incomplete list who is missing its tail, and a destination pointing to the missing tail's location. The incomplete list is represented by the left side of the ampar, and the destination is represented by its right side. No read on the unfinished list can happen until the destination on the right has been consumed.

Type and operators definitions for difference lists in λ_d are presented on the left of Figure 2.

An empty difference list can be created without the need for a new operator as it is exactly what the alloc primitive already does when specialized to type $\mathsf{List}\ \mathsf{T}$.

The **append** operator uses "fillCons" to link a new hollow "cons" cell at the end of the list (represented by dys), and then handles the two associated destinations dy and dys'. The former, representing the item slot, is fed with the item to append, while the latter, representing the slot for the tail of the resulting difference list, is returned and so stored back in the right side of the ampar. If that second destination was consumed, and not returned, we would end up with a regular linked list, instead of a difference list.

$$
\begin{array}{l}
\mathsf{DList\ T} \triangleq (\mathsf{List\ T}) \ltimes \lfloor \mathsf{List\ T} \rfloor \\[4pt]
\textbf{append} : \mathsf{DList\ T} \to \mathsf{T} \to \mathsf{DList\ T} \\
ys\ \textbf{append}\ y \triangleq \\
\quad \textbf{map}\ ys\ \textbf{with}\ dys \mapsto \textbf{case}\ (dys \triangleleft (::))\ \textbf{of} \\
\qquad (dy, dys') \mapsto dy \blacktriangleleft y \fatsemi dys' \\[4pt]
\textbf{concat} : \mathsf{DList\ T} \to \mathsf{DList\ T} \to \mathsf{DList\ T} \\
ys\ \textbf{concat}\ ys' \triangleq \textbf{map}\ ys\ \textbf{with}\ d \mapsto d \triangleleft\!\bullet ys' \\[4pt]
\textbf{to}_{\textsf{List}} : \mathsf{DList\ T} \to \mathsf{List\ T} \\
\textbf{to}_{\textsf{List}}\ ys \triangleq \textbf{from}'_{\ltimes}(\textbf{map}\ ys\ \textbf{with}\ d \mapsto d \triangleleft [\,])
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Queue\ T} \triangleq (\mathsf{List\ T}) \otimes (\mathsf{DList\ T}) \\[4pt]
\textbf{singleton} : \mathsf{T} \to \mathsf{Queue\ T} \\
\textbf{singleton}\ x \triangleq (\mathsf{Inr}\,(x, \mathsf{Inl}\,()), \mathsf{alloc}) \\[4pt]
\textbf{enqueue} : \mathsf{Queue\ T} \to \mathsf{T} \to \mathsf{Queue\ T} \\
q\ \textbf{enqueue}\ y \triangleq \\
\quad \textbf{case}\ q\ \textbf{of}\ (xs, ys) \mapsto (xs, ys\ \textbf{append}\ y) \\[4pt]
\textbf{dequeue} : \mathsf{Queue\ T} \to 1 \oplus (\mathsf{T} \otimes (\mathsf{Queue\ T})) \\
\textbf{dequeue}\ q \triangleq \\
\quad \textbf{case}\ q\ \textbf{of}\ \{ \\
\qquad (\mathsf{Inr}\,(x, xs), ys) \mapsto \mathsf{Inr}\,(x, (xs, ys)), \\
\qquad (\mathsf{Inl}\,(), ys) \mapsto \textbf{case}\ (\textbf{to}_{\textsf{List}}\ ys)\ \textbf{of}\ \{ \\
\qquad\quad \mathsf{Inl}\,() \mapsto \mathsf{Inl}\,(), \\
\qquad\quad \mathsf{Inr}\,(x, xs) \mapsto \mathsf{Inr}\,(x, (xs, \mathsf{alloc})) \\
\qquad \} \\
\quad \}
\end{array}
$$

Fig. 2. Difference list and queue implementation in equirecursive λ_d

The **concat** operator concatenates two difference lists by writing the head of the second one to the hole at the end of the first one. This is done using the "fillComp" ($\triangleleft\!\bullet$) primitive, that here has type $\lfloor \mathsf{List\ T} \rfloor \to \mathsf{DList\ T}\ {}_{1\uparrow}\!\!\to \lfloor \mathsf{List\ T} \rfloor$. It takes a destination as a first argument and an ampar as its second argument. The left side of the ampar is fed to the destination to be merged with another incomplete structure (the first difference list), while right side of the ampar (the destination for the end of the second difference list) is returned.

Finally, the $\textbf{to}_{\textsf{List}}$ operator converts a difference list to a regular list by writing the "nil" constructor to the hole left in the incomplete list using "fillNil".

We can note that although this exemple is typical of destination-style programming, it doesn't use the first-class nature of destinations that our calculus allows, and thus can be implemented in other destination-passing style frameworks such as [Bour et al. 2021] and [Leijen and Lorenzen 2023]. We will see in the next sections what kind of programs can be benefit from first-class destinations.

Efficient queue using previously defined structures. The usual functional encoding for a queue is to use a pair of lists, one representing the front of the queue, and keeping the element in order, while the second list represent the back of the queue, and is kept in reverse order (e.g the latest inserted element will be at the front of the second list).

With such a queue implementation, dequeueing the front element is efficient until the first list is depleted. In that case, one now has to transfer elements from the second list to the first one, and so has to reverse the second list beforehand, which is a $O(n)$ operation (although it is amortized).

With access to efficient difference lists, as shown in the previous paragraph, we can replace the second list by a difference list, to remove the need for a reverse operation when the first list get depleted. The corresponding implementation is presented on the left of Figure 2.

The **singleton** operator creates a pair of a list with a single element, and a fresh difference list (obtained via alloc).

The **enqueue** operator appends an element to the difference list, while letting the front list unchanged.

The **dequeue** operator first checks if there is one element available in the front list. If there is, it extracts the element x and returns it alongside the rest of the queue (xs, ys). If there isn't, it converts the difference list *ys* to a normal list, and pattern-matches on it to look for an available element. If none is found again, it returns $\mathsf{Inl}\ ()$ to signal that the queue is definitely empty. If an element x is found, then it returns it alongside the updated queue (made of the remaining elements *xs* paired with a fresh difference list).

3 SCOPE ESCAPE OF DESTINATIONS

Everything described in Section 2 is in fact already possible in destination-passing style for Haskell as presented in [Bagrel 2024]. However, in the aforementioned paper, destinations cannot be stored in destination-based data structures. This restriction is a rather blunt approach to prevent scope escape of destinations that itself pose a great threat to the memory safety of the system. Let's see why.

One core assumption of destination-passing style programming is that once a destination has been linearly used, the associated hole has been written to. However, in a realm where destinations $\lfloor \mathsf{T} \rfloor$ can be of arbitrary inner type T, they can in particular be used to store a destination itself when $\mathsf{T} = \lfloor \mathsf{T}' \rfloor$!

The value fed in a destination is linearly used, which means it cannot be both stored away to be used later, and used in the current context (as that would result in two uses). But that also means that the destination $d : \lfloor \mathsf{T}' \rfloor$ is linearly used too when it is fed to $dd : \lfloor \lfloor \mathsf{T}' \rfloor \rfloor$ in $dd \blacktriangleleft d$.

As a result, there are in fact two ways to use a destination linearly: fill it now with a value, or store it away and fill it later. The latter is a much weaker form of linear use, as it doesn't guarantee that the hole associated to the destination has been written to *now*, only that it will be written to later. So our initial assumption above doesn't hold in general case.

The issue is particularly visible when trying to give semantics to the **alloc′** operator:

the mode on the rightmost arrow should be 1∞, but do we want to show that now?

$$\begin{aligned} \mathbf{alloc}' &: (\lfloor \mathsf{T} \rfloor \to 1) \to \mathsf{T} \\ \mathbf{alloc}'\ f &\triangleq \mathbf{from}'_{\ltimes} (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto f\ d) \end{aligned}$$

With linear store semantics, this is how **alloc′** would behave:

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

I'm [Arnaud] really unsure about introducing this one-off notation that we're never using again. Even though I like the use of the underwave to highlight the problem.

It works as expected when the function supplied to **alloc′** will indeed use the destination to store a value:

$$\begin{array}{rrcl} & \{\} & \mid & \mathbf{alloc}'\ (\lambda d \mapsto d \triangleleft \mathsf{Inl} \triangleleft ()) \\ \longrightarrow & \{h {:=} \square\} & \mid & {\to}h \triangleleft \mathsf{Inl} \triangleleft () \mathbin{⨟} \mathbf{deref}\ {\to}h \\ \longrightarrow & \{h {:=} \mathsf{Inl}\ ()\} & \mid & \mathbf{deref}\ {\to}h \\ \longrightarrow & \{\} & \mid & \mathsf{Inl}\ () \end{array}$$

However this falls short when calls to **alloc′** are nested in the following way (where $dd : \lfloor \lfloor 1 \rfloor \rfloor$ and $d : \lfloor 1 \rfloor$):

$$\begin{array}{rrcl} & \{\} & \mid & \mathbf{alloc}'\ (\lambda dd \mapsto \mathbf{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d)) \\ \longrightarrow & \{hd {:=} \square\} & \mid & \mathbf{alloc}'\ (\lambda d \mapsto {\to}hd \blacktriangleleft d) \mathbin{⨟} \mathbf{deref}\ {\to}hd \\ \longrightarrow & \{hd {:=} \square, h {:=} \square\} & \mid & {\to}hd \blacktriangleleft {\to}h \mathbin{⨟} \mathbf{deref}\ {\to}h \mathbin{⨟} \mathbf{deref}\ {\to}hd \\ \longrightarrow & \{hd {:=} {\to}h, h {:=} \square\} & \mid & \mathbf{deref}\ {\to}h \mathbin{⨟} \mathbf{deref}\ {\to}hd \end{array}$$

The original term $\mathbf{alloc}'\ (\lambda dd \mapsto \mathbf{alloc}'\ (\lambda d \mapsto dd \blacktriangleleft d))$ is well typed, as the inner call to **alloc′** returns a value of type 1 (as d is of type $\lfloor 1 \rfloor$) and uses d linearly. However, the variable d that stands for destination ${\to}h$ isn't filled with a value but instead escapes its scope by being fed to a destination of destination dd coming from the outer scope (this still counts as a linear usage). Hence the associated hole h doesn't receive a value and the reduction get stuck when trying to dereference ${\to}h$.

I remember that we did wonder about that at some point, so it's worth pre-empting the question. Though I'd rather we found a slightly diffe

One could argue that the issue comes from the primitive $\blacktriangleleft$ returning a value of type 1 instead of a dedicated effect type. However, the same issue arise in following program, which is well-typed even when $\blacktriangleleft$ and the function accepted by $\mathbf{alloc'}$ have an arbitrary return type E, so we decided not to introduce yet another type in the system:

$$\begin{aligned}&\mathbf{alloc'}\ (\lambda dd \mapsto \mathbf{case}\ (dd \triangleleft (,))\ \mathbf{of}\ (dd_1, d_2) \mapsto \\ &\qquad\qquad \mathbf{case}\ (\mathbf{alloc'}\ (\lambda d \mapsto dd_1 \blacktriangleleft d))\ \mathbf{of}\ \{\mathsf{true} \mapsto d_2 \blacktriangleleft \mathsf{true},\ \mathsf{false} \mapsto d_2 \blacktriangleleft \mathsf{false}\})\end{aligned}$$

where $dd : \lfloor\lfloor\mathsf{Bool}\rfloor \otimes \mathsf{Bool}\rfloor$, $dd_1 : \lfloor\lfloor\mathsf{Bool}\rfloor\rfloor$, $d_2 : \lfloor\mathsf{Bool}\rfloor$, $d : \lfloor\mathsf{Bool}\rfloor$.

In the next section, we motivate why being able to store destinations in destinations of destinations is a desirable property of our system that we don't want to give up. In Section 5, we present a finer type system that prevents scope escape while still allowing to store destinations in destination-based data structures.

4 BREADTH-FIRST TREE TRAVERSAL

The core example that showcases the power of destination-passing style programming with first-class destination — that we borrow from [Bagrel 2024] — is breadth-first tree traversal:

> 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.

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.

In the aforementioned paper, the author presents a breadth-first traversal implementation that relies on first-class destinations so as to build the final tree in a single pass over the input tree. Their implementation, exactly like ours, uses a queue to store pairs of an input subtree and a destination to the corresponding output subtree. This queue is what materialize the breadth-first processing order: the leading pair (*input subtree*, *dest to output subtree*) of the queue is processed, and pairs of the same shape for children nodes are appended at the end of the queue.

Rework the next couple of paragraph to flow a little bit better.

However, as evoked earlier, the API presented in [Bagrel 2024] is not able to store linear data, and in particular destinations, in destination-based data structures. So they cannot use the efficient, destination-based queue implementation from Section 2.2 to power up the breadth-first tree traversal implementation[2]. With λ_d , this is now possible. In fact, our system is self-contained, in the sense that every possible structure can be built using destination-based primitives (and regular data constructors can be retrieved from destination-based primitives, as detailed in Figure 5).

Figure 3 presents the λ_d implementation of the breadth-first tree traversal. We assume that we have a binary tree type alias $\mathsf{Tree\ T}$ and natural number type alias Nat encoded using standard sum and product types. $\mathsf{Tree\ T}$ is equipped with operators $\triangleleft\mathsf{Nil}$ and $\triangleleft\mathsf{Node}$, that are implemented in terms of our core destination-filling primitives.

The stateful transformer f that is applied to each input node has type $\mathsf{S}\ {}_{1\infty}\!\!\to \mathsf{T}_1 \to (!_{1\infty}\mathsf{S}) \otimes \mathsf{T}_2$. It takes the current state and node value and returns the next state and value for output node. The state has to be wrapped in an exponential $!_{1\infty}$ in the return type to witness that it cannot capture destinations. That way, the state can be extracted using $\mathbf{from}_\ltimes$ at the end of the processing.

The $\mathbf{go}$ function is in charge of consuming the queue containing the pairs of input subtrees and destinations to the corresponding output subtrees. It dequeues the first pair, and processes it. If the input subtree is Nil, it fills Nil into the destination for the output tree and continues the processing of next elements with unchanged state. If the input subtree is a node, it writes a hollow Node constructor to the hole pointed to by the destination *dtree*, processes the value x of the input node with the stateful transformer f, and continues the processing of the updated queue where children subtrees and their associated destinations have been enqueued.

[2] This efficient queue implementation can be, and is in fact, implemented in [Bagrel 2024]: see archive.softwareheritage.org/swh:1:cnt:29e9d1fd48d94fa8503023bee0d607d281f512f8. But it cannot store linear data

```
go : (S 1∞⟶ T₁ ⟶ (!1∞S)⊗T₂) ω∞⟶ S 1∞⟶ Queue (Tree T₁⊗⌊Tree T₂⌋) ⟶ (!1∞S)
go f st q ≜rec case (dequeue q) of {
            Inl () ↦ E1∞ st ,
            Inr ((tree, dtree), q′) ↦ case tree of {
                Inl () ↦ dtree ◂ Nil ⨾ go f st q′ ,
                Inr (x, (tl, tr)) ↦ case (dtree ◂ Node) of
                    (dy, (dtl, dtr)) ↦ case (f st x) of
                        (E1∞ st′ , y) ↦
                            dy ◀ y ⨾
                            go f st′ (q′ enqueue (tl, dtl) enqueue (tr, dtr))
            }
        }

mapAccumBFS : (S 1∞⟶ T₁ ⟶ (!1∞S)⊗T₂) ω∞⟶ S 1∞⟶ Tree T₁ 1∞⟶ Tree T₂⊗(!1∞S)
mapAccumBFS f st tree ≜ from⋉ (map alloc with dtree ↦ go f st (singleton (tree, dtree)))

relabelDPS : Tree 1 1∞⟶ (Tree Nat)⊗(!1∞(!ων Nat))
relabelDPS tree ≜ mapAccumBFS
                    (λex 1∞↦ λun ↦ un ⨾ case1∞ ex of
                        Eων st ↦ (E1∞ (Eων (succ st)), st))
                    (Eων (succ zero))
                    tree
```

Fig. 3. Breadth-first tree traversal in destination-passing style

mapAccumBFS spawns the initial memory slot for the output tree, and prepares the initial queue containing a single pair, made of the whole input tree and a destination to the aforementioned memory slot.

relabelDPS is a special case of **mapAccumBFS** that takes the skeleton of a tree (where node values are all unit) and returns a tree of integers, with the same skeleton, but with node values replaced by naturals $1 \ldots |T|$ in breadth-first order. The higher-order function passed to **mapAccumBFS** is quite verbose: it must consume the value of the input node (unit) using $\fatsemi$, then extract the state (representing the next natural number to attribute to a node) from its exponential wrapper, and finally return a pair, whose left side is the incremented natural wrapped back into its two exponential layers (new label for next node), and whose right side is the original natural acting as a label for the current node The extra exponential $!_{\omega\nu}$ around Nat let us use the natural number twice.

You might wonder what all the fuchsia subscripts 1∞, $\omega\nu$ . . . means. It's now time to cover the type and mode system of λ_d .

I'm [Arnaud] honestly uncomfortable at the quantity of mathematics written in this section without macros and completely uncorrelated from OTT. If we change any notation, we'll have to trawl through a lot of text to make sure they are properly propagated. Thomas: It's hard to al-

5 LANGUAGE SYNTAX AND TYPE SYSTEM

λ_d is based on simply typed lambda calculus, with a first-order type system, featuring modal function types and modal boxing, in addition to unit (1), product ($\otimes$) and sum ($\oplus$) types. It is also equipped with the destination type $\lfloor_m \mathsf{T}\rfloor$ and ampar type $\mathsf{S} \ltimes \mathsf{T}$ that have been previewed in Section 2 to represent DPS structure building. The core grammar of the language is presented in Figure 4. We also provide commonly used syntactic sugar forms for terms in Figure 5.

Modes in λ_d have two axes — multiplicity (i.e. linear/non-linear), and age control — and they take place on variable bindings in typing contexts Ω, and on function arrows, but are not part of the type itself.

$$
\begin{array}{rcl}
\mathrm{t}, \mathrm{u} & ::= & \mathrm{v} \mid x \mid \mathrm{t}'\ \mathrm{t} \mid \mathrm{t} \operatorname{⨟} \mathrm{t}' \\
& \mid & \mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ \{\mathsf{Inl}\, x_1 \mapsto \mathrm{u}_1\,,\ \mathsf{Inr}\, x_2 \mapsto \mathrm{u}_2\} \mid \mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ (x_1\,, x_2) \mapsto \mathrm{u} \mid \mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ \mathsf{E}_{\mathrm{n}}\, x \mapsto \mathrm{u} \\
& \mid & \mathbf{map}\ \mathrm{t}\ \mathbf{with}\ x \mapsto \mathrm{t}' \mid \mathbf{to}_{\ltimes}\ \mathrm{t} \mid \mathbf{from}_{\ltimes}\ \mathrm{t} \\
& \mid & \mathrm{t} \triangleleft () \mid \mathrm{t} \triangleleft \mathsf{Inl} \mid \mathrm{t} \triangleleft \mathsf{Inr} \mid \mathrm{t} \triangleleft (,) \mid \mathrm{t} \triangleleft \mathsf{E}_{\mathrm{m}} \mid \mathrm{t} \triangleleft (\lambda x\ {}_{\mathrm{m}}\!\mapsto \mathrm{u}) \mid \mathrm{t} \triangleleft\bullet \mathrm{t}' \\
\\
\mathrm{v} & ::= & \boxed{h} \qquad \textit{(hole)} \\
& \mid & \rightarrow h \qquad \textit{(destination)} \\
& \mid & {}_{H}\langle \mathrm{v}_2 \,{}_{\wedge}\, \mathrm{v}_1 \rangle \qquad \textit{(ampar value form)} \\
& \mid & () \mid {}^{\mathrm{v}}\lambda x\ {}_{\mathrm{m}}\!\mapsto \mathrm{u} \mid \mathsf{Inl}\, \mathrm{v} \mid \mathsf{Inr}\, \mathrm{v} \mid \mathsf{E}_{\mathrm{m}}\, \mathrm{v} \mid (\mathrm{v}_1\,, \mathrm{v}_2) \\
\\
\mathsf{T}, \mathsf{U}, \mathsf{S} & ::= & \lfloor_{\mathrm{m}} \mathsf{T} \rfloor \qquad \textit{(destination)} \\
& \mid & \mathsf{U} \ltimes \mathsf{T} \qquad \textit{(ampar)} \\
& \mid & 1 \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid !_{\mathrm{m}} \mathsf{T} \mid \mathsf{T}\ {}_{\mathrm{m}}\!\!\rightarrow \mathsf{U} \\
\\
\mathrm{m}, \mathrm{n} & ::= & \mathrm{pa} \qquad \textit{(pair of multiplicity and age)} \\
\mathrm{p} & ::= & 1 \mid \omega \\
\mathrm{a} & ::= & \nu \mid \uparrow \mid \infty \\
\\
\Omega, \Gamma, \Theta, \Delta & ::= & \bullet \mid x :_{\mathrm{m}} \mathsf{T} \mid \boxed{h} :_{\mathrm{n}} \mathsf{T} \mid \rightarrow h :_{\mathrm{m}} \lfloor_{\mathrm{n}} \mathsf{T} \rfloor \\
& \mid & \Omega_1,\ \Omega_2 \mid \Omega_1 + \Omega_2 \mid \mathrm{m}\cdot\Omega \mid \rightarrow^{-1}\Delta
\end{array}
$$

Fig. 4. Grammar of λ_d

$$
\begin{array}{l}
\mathsf{alloc} \triangleq {}_{\{1\}}\langle \boxed{1} \,{}_{\wedge}\, \rightarrow 1 \rangle \\[4pt]
\mathrm{t} \blacktriangleleft \mathrm{t}' \triangleq \mathrm{t} \triangleleft\bullet (\mathbf{to}_{\ltimes}\ \mathrm{t}') \\[4pt]
\mathsf{Inl}\, \mathrm{t} \triangleq \mathbf{from}'_{\ltimes}\, (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad\qquad d \triangleleft \mathsf{Inl} \blacktriangleleft \mathrm{t} \\
\qquad)\\[4pt]
\mathsf{Inr}\, \mathrm{t} \triangleq \mathbf{from}'_{\ltimes}\, (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad\qquad d \triangleleft \mathsf{Inr} \blacktriangleleft \mathrm{t} \\
\qquad)\\[4pt]
\mathsf{E}_{\mathrm{m}}\ \mathrm{t} \triangleq \mathbf{from}'_{\ltimes}\, (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad\qquad d \triangleleft \mathsf{E}_{\mathrm{m}} \blacktriangleleft \mathrm{t} \\
\qquad)
\end{array}
\qquad
\begin{array}{l}
\mathbf{from}'_{\ltimes}\ \mathrm{t} \triangleq \\
\quad \mathbf{case}\ (\mathbf{from}_{\ltimes}\ (\mathbf{map}\ \mathrm{t}\ \mathbf{with}\ un \mapsto un \operatorname{⨟} \mathsf{E}_{1\infty}\ ()))\ \mathbf{of} \\
\qquad (st\,, ex) \mapsto \mathbf{case}\ ex\ \mathbf{of} \\
\qquad\quad \mathsf{E}_{1\infty}\ un \mapsto un \operatorname{⨟} st \\[4pt]
\lambda x\ {}_{\mathrm{m}}\!\mapsto \mathrm{u} \triangleq \mathbf{from}'_{\ltimes}\, (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad\qquad d \triangleleft (\lambda x\ {}_{\mathrm{m}}\!\mapsto \mathrm{u}) \\
\qquad\quad)\\[4pt]
(\mathrm{t}_1\,, \mathrm{t}_2) \triangleq \mathbf{from}'_{\ltimes}\, (\mathbf{map}\ \mathsf{alloc}\ \mathbf{with}\ d \mapsto \\
\qquad\qquad \mathbf{case}\ (d \triangleleft (,))\ \mathbf{of} \\
\qquad\qquad\quad (d_1\,, d_2) \mapsto d_1 \blacktriangleleft \mathrm{t}_1 \operatorname{⨟} d_2 \blacktriangleleft \mathrm{t}_2 \\
\qquad\quad)
\end{array}
$$

Fig. 5. Syntactic sugar forms for terms

We omit the mode annotation m on function arrows and destinations when the mode in question is the multiplicative neutral element 1ν of the mode semiring (in particular, a function arrow without annotation is linear by default). A function arrow with multiplicity 1 is equivalent to the linear arrow $\multimap$ from [Girard 1995].

Let's now introduce the age mode axis, which is a novel feature of this calculus.

5.1 Age-control for bindings to prevent scope escape of destinations

The solution we chose to alleviate scope escape of destinations (detailed in Section 3) is to track the age of destinations (as De-Brujin-like scope indices), and to set age-control restriction on the typing rule of destination-filling primitives.

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

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

$+$	1	ω
1	ω	ω
ω	ω	ω

$\cdot$	1	ω
1	1	∞
ω	∞	∞

We pose $\uparrow^0 = \nu$ and $\uparrow^n = \uparrow \cdot \uparrow^{n-1}$

Fig. 6. Operation tables for age and multiplicity semirings

Age is represented by a commutative semiring, where ν indicates that a destination originates from the current scope, and $\uparrow$ indicates that it originates from the scope just before. We also extend ages to variables (a variable of age a stands for a value of age a). Finally, age ∞ is introduced for variables standing in place of a non-age-controlled value. In particular, destinations can never have age ∞; a main role of age ∞ is thus to act as a proof that no destination can be part of the value.

Semiring addition $+$ is used to find the age of a variable or destination that is used in two subterms of a program. Semiring multiplication $\cdot$ corresponds to age composition, and is in fact an integer sum on scope indices. ∞ is absorbing for both addition and multiplication.

Tables for the operations $+$ and $\cdot$ are presented in Figure 6.

Age commutative semiring is then combined with the multiplicity commutative semiring from [Bernardy et al. 2018] to form a canonical product commutative semiring that forms the mode of each typing context binding in our final type system.

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 integrating the idea of incomplete data structures (structures with holes) that exist as first class values and can be interacted with by the user.

In that paper, 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, we know we can transform linear implication $\mathsf{T} \multimap \mathsf{S}$ into $\mathsf{S} \parr \mathsf{T}^{\perp}$. Doing so for the type $(\mathsf{T}, \mathsf{S})\mathsf{hfun}$ gives $\mathsf{S}\ \widehat{\parr}\ \lfloor \mathsf{T} \rfloor$, where $\lfloor \cdot \rfloor$ is memory negation, and $\widehat{\parr}$ is a memory *par* (it allows less interaction than the CLL *par*, because hfun is weaker than $\multimap$).

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

From memory par $\widehat{\parr}$ *to ampar* $\ltimes$. TODO: Should I mention that dests is a non-involutive negation?

In CLL, thanks to the cut rule, any of the sides S or T of a par $\mathsf{S} \parr \mathsf{T}$ can be eliminated, by interaction with the opposite type $\cdot^{\perp}$, to free up the other side. But in λ_d , we have two types of interaction to consider: interaction between T and $\lfloor \mathsf{T} \rfloor$, and interaction between T and $\mathsf{T} \rightarrow \cdot$. The structure that may contain holes, S, can safely interact with $\lfloor \mathsf{S} \rfloor$ (merge it into a bigger structure with holes), but not with $\mathsf{T} \rightarrow \cdot$, as it would let the user 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 $\mathsf{T} \rightarrow \mathsf{1}$ (in particular, the function can pattern-match on the value of type T to access the destinations), but it is not always safe to fill it into a $\lfloor \mathsf{T} \rfloor$ as that might allow scope escape of destination $\lfloor \mathsf{T}' \rfloor$ as we've just seen in Section 3.

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

- the left side S can contain holes, and can be only be eliminated by interaction with $\lfloor S \rfloor$ using "fillComp" ($\triangleleft\!\bullet$) to free up the right side T;
- the right side T cannot contain holes (it might contain destinations), and can be eliminated by interaction with $T \multimap 1$ to free up the left side S. At term level, this is done using $\mathbf{from}'_{\ltimes}$ and **map**.

5.3 Typing of terms and values

The typing rules for λ_d are highly inspired from[Abel and Bernardy 2020] and Linear Haskell [Bernardy et al. 2018], and are detailed in Figure 7. In particular, we use the same additive/multiplicative approach on contexts for linearity and age enforcement

Destinations and holes are two faces of the same coin, as seen in Section 2.1, and must always be in 1:1 correspondance. Thus, the new idea of our type system is to feature *hole bindings* $\boxed{h} :_{n} T$ and *destination bindings* $\rightarrow\!h :_{m} \lfloor_{n} T \rfloor$ in addition to the variable bindings $x :_{m} T$ that usually populates typing contexts.

Such bindings mention two distinct classes of names: regular variable names x, y, and *hole names* h, h_1, h_2 which are identifiers for a memory cell that hasn't been written to yet. Hole names are represented by natural numbers under the hood, so they are equipped with addition $h{+}h'$ and can act both as relative offsets or absolute positions in memory. Typically, when a structure is effectively allocated, its hole (and destination) names are shifted by the maximum hole name encountered so far in the program (denoted $\mathbf{max}(\mathbf{hnames}(C))$) ; this corresponds to finding the next unused memory cell in which to write new data.

The mode n of a hole binding $\boxed{h} :_{n} T$ (also present in the corresponding destination type $\lfloor_{n} T \rfloor$) indicates the mode a value must have to be written to it (that is to say, the mode of bindings that the value depends on to type correctly).[3] We see the mode of a hole coming into play when a hole is located behind an exponential constructor: we should only write a non-linear value to the hole $\boxed{h}$ in $\mathsf{E}_{\omega\nu}\,\boxed{h}$. In particular, we should not store a destination into this hole, otherwise it could later be extracted and used in a non-linear fashion.

A destination binding $\rightarrow\!h :_{m} \lfloor_{n} T \rfloor$ mentions two modes, m and n; but only the former (left one, m) is the actual mode of the binding (in particular, it informs on the age of the destination itself). The latter, n, is part of the destination's type $\lfloor_{n} T \rfloor$ and corresponds to the mode a value has to have to be written to the corresponding hole. In a well-typed, closed program, the mode m of a destination binding can never be of multiplicity ω or age ∞ in the typing tree; it is always linear and of finite age.

We also extend mode product to a point-wise action on typing contexts:

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

Figure 7 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. In every figure,

- Ω denotes an arbitrary typing context, with no particular constraint;

[3]To this day, the only way for a value to have a constraining mode is to capture a destination (otherwise the value has mode $\omega\infty$, meaning it can be used in any possible way), as destinations are the only intrinsically linear values in the calculus, but we will see in Section 8 that other forms of intrinsic linearity can be added to the langage for practical reasons.

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

$$\textsc{Ty-term-Val}\quad \frac{\mathtt{DisposableOnly}\ \Theta \qquad \Delta \ {}^{\mathsf{v}}\!\vdash \mathrm{v} : \mathrm{T}}{\Theta,\ \Delta \vdash \mathrm{v} : \mathrm{T}}
\qquad
\textsc{Ty-term-Var}\quad \frac{\mathtt{DisposableOnly}\ \Theta \qquad 1\nu <: \mathrm{m}}{\Theta,\ x :_{\mathrm{m}} \mathrm{T} \vdash x : \mathrm{T}}
\qquad
\textsc{Ty-term-App}\quad \frac{\Theta_1 \vdash \mathrm{t} : \mathrm{T} \qquad \Theta_2 \vdash \mathrm{t}' : \mathrm{T}\ {}_{\mathrm{m}}\!\!\to \mathrm{U}}{\mathrm{m}\cdot\Theta_1 + \Theta_2 \vdash \mathrm{t}'\ \mathrm{t} : \mathrm{U}}$$

$$\textsc{Ty-term-PatU}\quad \frac{\Theta_1 \vdash \mathrm{t} : 1 \qquad \Theta_2 \vdash \mathrm{u} : \mathrm{U}}{\Theta_1 + \Theta_2 \vdash \mathrm{t}\ ⨾\ \mathrm{u} : \mathrm{U}}
\qquad
\textsc{Ty-term-PatS}\quad \frac{\Theta_1 \vdash \mathrm{t} : \mathrm{T}_1 \oplus \mathrm{T}_2 \qquad \Theta_2,\ x_1 :_{\mathrm{m}} \mathrm{T}_1 \vdash \mathrm{u}_1 : \mathrm{U} \qquad \Theta_2,\ x_2 :_{\mathrm{m}} \mathrm{T}_2 \vdash \mathrm{u}_2 : \mathrm{U}}{\mathrm{m}\cdot\Theta_1 + \Theta_2 \vdash \mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathrm{u}_1,\ \mathsf{Inr}\ x_2 \mapsto \mathrm{u}_2\} : \mathrm{U}}$$

$$\textsc{Ty-term-PatP}\quad \frac{\Theta_1 \vdash \mathrm{t} : \mathrm{T}_1 \otimes \mathrm{T}_2 \qquad \Theta_2,\ x_1 :_{\mathrm{m}} \mathrm{T}_1,\ x_2 :_{\mathrm{m}} \mathrm{T}_2 \vdash \mathrm{u} : \mathrm{U}}{\mathrm{m}\cdot\Theta_1 + \Theta_2 \vdash \mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ (x_1, x_2) \mapsto \mathrm{u} : \mathrm{U}}
\qquad
\textsc{Ty-term-PatE}\quad \frac{\Theta_1 \vdash \mathrm{t} : !_{\mathrm{n}}\mathrm{T} \qquad \Theta_2,\ x :_{\mathrm{m}\cdot\mathrm{n}} \mathrm{T} \vdash \mathrm{u} : \mathrm{U}}{\mathrm{m}\cdot\Theta_1 + \Theta_2 \vdash \mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ \mathsf{E}_{\mathrm{n}}\ x \mapsto \mathrm{u} : \mathrm{U}}$$

$$\textsc{Ty-term-Map}\quad \frac{\Theta_1 \vdash \mathrm{t} : \mathrm{U} \ltimes \mathrm{T} \qquad 1\!\uparrow\!\cdot\Theta_2,\ x :_{1\nu} \mathrm{T} \vdash \mathrm{t}' : \mathrm{T}'}{\Theta_1 + \Theta_2 \vdash \mathbf{map}\ \mathrm{t}\ \mathbf{with}\ x \mapsto \mathrm{t}' : \mathrm{U} \ltimes \mathrm{T}'}
\qquad
\textsc{Ty-term-ToA}\quad \frac{\Theta \vdash \mathrm{u} : \mathrm{U}}{\Theta \vdash \mathbf{to}_{\ltimes}\ \mathrm{u} : \mathrm{U} \ltimes 1}
\qquad
\textsc{Ty-term-FromA}\quad \frac{\Theta \vdash \mathrm{t} : \mathrm{U} \ltimes (!_{1\infty}\mathrm{T})}{\Theta \vdash \mathbf{from}_{\ltimes}\ \mathrm{t} : \mathrm{U} \otimes (!_{1\infty}\mathrm{T})}$$

$$\textsc{Ty-term-FillU}\quad \frac{\Theta \vdash \mathrm{t} : \lfloor_{\mathrm{n}} 1 \rfloor}{\Theta \vdash \mathrm{t} \triangleleft () : 1}
\qquad
\textsc{Ty-term-FillL}\quad \frac{\Theta \vdash \mathrm{t} : \lfloor_{\mathrm{n}} \mathrm{T}_1 \oplus \mathrm{T}_2 \rfloor}{\Theta \vdash \mathrm{t} \triangleleft \mathsf{Inl} : \lfloor_{\mathrm{n}} \mathrm{T}_1 \rfloor}
\qquad
\textsc{Ty-term-FillR}\quad \frac{\Theta \vdash \mathrm{t} : \lfloor_{\mathrm{n}} \mathrm{T}_1 \oplus \mathrm{T}_2 \rfloor}{\Theta \vdash \mathrm{t} \triangleleft \mathsf{Inr} : \lfloor_{\mathrm{n}} \mathrm{T}_2 \rfloor}
\qquad
\textsc{Ty-term-FillP}\quad \frac{\Theta \vdash \mathrm{t} : \lfloor_{\mathrm{n}} \mathrm{T}_1 \otimes \mathrm{T}_2 \rfloor}{\Theta \vdash \mathrm{t} \triangleleft (,) : \lfloor_{\mathrm{n}} \mathrm{T}_1 \rfloor \otimes \lfloor_{\mathrm{n}} \mathrm{T}_2 \rfloor}$$

$$\textsc{Ty-term-FillE}\quad \frac{\Theta \vdash \mathrm{t} : \lfloor_{\mathrm{n}} !_{\mathrm{n}'}\mathrm{T} \rfloor}{\Theta \vdash \mathrm{t} \triangleleft \mathsf{E}_{\mathrm{n}'} : \lfloor_{\mathrm{n}'\cdot\mathrm{n}} \mathrm{T} \rfloor}
\qquad
\textsc{Ty-term-FillF}\quad \frac{\Theta_1 \vdash \mathrm{t} : \lfloor_{\mathrm{n}} \mathrm{T}\ {}_{\mathrm{m}}\!\!\to \mathrm{U} \rfloor \qquad \Theta_2,\ x :_{\mathrm{m}} \mathrm{T} \vdash \mathrm{u} : \mathrm{U}}{\Theta_1 + (1\!\uparrow\!\cdot\mathrm{n})\cdot\Theta_2 \vdash \mathrm{t} \triangleleft (\lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u}) : 1}
\qquad
\textsc{Ty-term-FillComp}\quad \frac{\Theta_1 \vdash \mathrm{t} : \lfloor_{\mathrm{n}} \mathrm{U} \rfloor \qquad \Theta_2 \vdash \mathrm{t}' : \mathrm{U} \ltimes \mathrm{T}}{\Theta_1 + (1\!\uparrow\!\cdot\mathrm{n})\cdot\Theta_2 \vdash \mathrm{t} \triangleleft\!\bullet\ \mathrm{t}' : \mathrm{T}}$$

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

$$\textsc{Ty-sterm-Alloc}\quad \frac{\mathtt{DisposableOnly}\ \Theta}{\Theta \vdash \mathsf{alloc} : \mathrm{T} \ltimes \lfloor \mathrm{T} \rfloor}
\qquad
\textsc{Ty-sterm-FromA'}\quad \frac{\Theta \vdash \mathrm{t} : \mathrm{T} \ltimes 1}{\Theta \vdash \mathbf{from}'_{\ltimes}\ \mathrm{t} : \mathrm{T}}
\qquad
\textsc{Ty-sterm-FillLeaf}\quad \frac{\Theta_1 \vdash \mathrm{t} : \lfloor_{\mathrm{n}} \mathrm{T} \rfloor \qquad \Theta_2 \vdash \mathrm{t}' : \mathrm{T}}{\Theta_1 + (1\!\uparrow\!\cdot\mathrm{n})\cdot\Theta_2 \vdash \mathrm{t} \blacktriangleleft \mathrm{t}' : 1}$$

$$\textsc{Ty-sterm-Fun}\quad \frac{\Theta_2,\ x :_{\mathrm{m}} \mathrm{T} \vdash \mathrm{u} : \mathrm{U}}{\Theta_2 \vdash \lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u} : \mathrm{T}\ {}_{\mathrm{m}}\!\!\to \mathrm{U}}
\qquad
\textsc{Ty-sterm-Left}\quad \frac{\Theta_2 \vdash \mathrm{t} : \mathrm{T}_1}{\Theta_2 \vdash \mathsf{Inl}\ \mathrm{t} : \mathrm{T}_1 \oplus \mathrm{T}_2}
\qquad
\textsc{Ty-sterm-Right}\quad \frac{\Theta_2 \vdash \mathrm{t} : \mathrm{T}_2}{\Theta_2 \vdash \mathsf{Inr}\ \mathrm{t} : \mathrm{T}_1 \oplus \mathrm{T}_2}
\qquad
\textsc{Ty-sterm-Exp}\quad \frac{\Theta_2 \vdash \mathrm{t} : \mathrm{T}}{\mathrm{m}\cdot\Theta_2 \vdash \mathsf{E}_{\mathrm{m}}\ \mathrm{t} : !_{\mathrm{m}}\mathrm{T}}$$

$$\textsc{Ty-sterm-Prod}\quad \frac{\Theta_{21} \vdash \mathrm{t}_1 : \mathrm{T}_1 \qquad \Theta_{22} \vdash \mathrm{t}_2 : \mathrm{T}_2}{\Theta_{21} + \Theta_{22} \vdash (\mathrm{t}_1, \mathrm{t}_2) : \mathrm{T}_1 \otimes \mathrm{T}_2}$$

Fig. 7. Typing rules for terms and syntactic sugar

- Γ denotes a typing context made only of hole and destination bindings;
- Θ denotes a typing context made only of destination and variable bindings;
- Δ denotes a typing context made only of destination bindings.

5.3.1 Typing of terms $\vdash$. A term t always types in a context Θ made only of destination and variable bindings. That being said, typing rules for terms and their syntactic sugar in Figure 7 never explicitly mention a destination binding; only variable bindings. Only at runtime some variables will be substituted by destinations having a matching type and mode (that is why terms cannot type in a context made of variable bindings alone). As a result, the type system the user has to deal with is only slightly more complex than a linear type system *à la* [Bernardy et al. 2018], because of the addition of the age control axis. So at the moment, we can forget about destination and hole bindings specificities.

Let's focus on a few particularities of the type system for terms.

The predicate `DisposableOnly` Θ in rules Ty-term-Val and Ty-term-Var says that Θ can only contain variable bindings with multiplicity ω, for which weakening is allowed in linear logic. It is enough to allow weakening at the leaves of the typing tree, that is to say the two aforementioned rules (Ty-term-Val is indeed a leaf for judgments $\vdash$, that holds a subtree of judgments ${}^{\mathsf{v}}\!\vdash$).

revisit if we allow weakening for dests, as we could replace that by $\omega\nu\cdot\Theta$ as in [Bernardy et al. 2018]

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 $pa <: p'a' \iff p \overset{\mathbf{p}}{<:} p' \wedge a \overset{\mathbf{a}}{<:} a'$ where:

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

Rule Ty-term-PatU is elimination (or pattern-matching) for unit, and is also used to chain destination-filling operations.

Pattern-matching with rules Ty-term-App, Ty-term-PatS, Ty-term-PatP and Ty-term-PatE is parametrized by a mode m by which the typing context Θ_1 of the scrutinee is multiplied. The variables which bind the subcomponents of the scrutinee then inherit this mode. In particular, this choice crystalize 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]. We omit the mode annotation on **case** statements and lambda abstractions when the mode in question is the multiplicative neutral element 1ν of the mode semiring.

The Rule Ty-term-Map is where most of the safety of the system lies, and it is there where scope control takes place. It opens an ampar t, and binds its right side (containing destinations for holes on the other side, among other things) to variable x and then execute body t'. This lets the user access destinations of an ampar while temporarily forgetting about the structure with holes (being mutated behind the scenes by the destination-filling primitives). Type safety for **map** is based on the idea that a new scope is created for x and t', so anything already present in the ambient scope (represented by Θ_2 in the conclusion) appears older when we see it from t' point of view. Indeed, when entering a new scope, the age of every remaining binding from the previous scopes is incremented by $\uparrow$. That way we can distinguish x from anything else that was already bound using the age of bindings alone. That's why t' types in $1\uparrow\cdot\Theta_2,\ x :_{1\nu} T$ while the global term **map** t **with** $x \mapsto t'$ types in $\Theta_1,\ \Theta_2$ (notice the absence of shift on Θ_2). A schematic explanation of the scope rules is given in Figure 8.

We see in the schema that the left of an ampar (the structure being built) "takes place" in the ambient scope. The right side however, where destinations are, has its own new, inner scope that is opened when **map**ped over. When filling a destination (e.g. $x_1 \blacktriangleleft x_0$ in the figure), the right operand must be from a scope $\uparrow$ older than the destination on the left of the operator, as this value will end up on the left of the ampar (which is thus in a scope $\uparrow$ older than the destination originating from the right side).

The rule Ty-term-FillComp, or its simpler variant, Ty-sterm-FillLeaf from Figure 7 confirm this intuition. The left operand of these operators must be a destination that types in the ambient

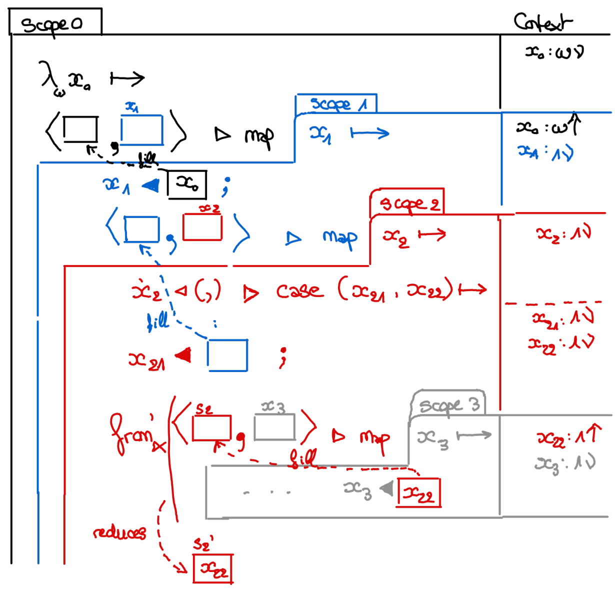

Fig. 8. Scope rules for **map** in λ_d

context (both Θ_1 unchanged in the premise and conclusion of the rules). The right operand, however, is a value that types in a context Θ_2 in the premise, but requires $1{\uparrow}{\cdot}\Theta_2$ in the conclusion. This is the opposite of the shift that **map** does: while **map** opens a child scope for its body, "fillComp" ($\triangleleft\bullet$)/"fillLeaf"($\blacktriangleleft$) opens a portal to the parent scope for their right operand, as seen in the schema. The same phenomenon happens for the resources captured by the body of a lambda abstraction in Ty-term-FillF.

In Ty-term-ToA, the operator $\mathbf{to}_{\ltimes}$ embeds an already completed structure in an ampar whose left side is the structure, and right side is unit.

When using $\mathbf{from}'_{\ltimes}$ (rule Ty-sterm-FromA'), the left of an ampar is extracted to the ambient scope (as seen at the bottom of Figure 8 with x_{22}): this is the fundamental reason why the left of an ampar has to "take place" in the ambient scope. We know the structure is complete and can be extracted because the right side is of type unit (1), and thus no destination on the right side means no hole can remain on the left. $\mathbf{from}'_{\ltimes}$ is implemented in terms of $\mathbf{from}_{\ltimes}$ in Figure 5 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 ambient scope, even though the right side is normally in a different scope. This is only safe to do because the right side is required to have type $!_{1\infty}\mathsf{T}$, which means it is scope-insensitive (it cannot contain any scope-controlled resource). This also ensures that the right side cannot contain destinations, meaning that the structure on the left is complete and ready to be read.

The remaining operators $\triangleleft()$, $\triangleleft\mathsf{Inl}$, $\triangleleft\mathsf{Inr}$, $\triangleleft\,\mathsf{E}_{m}$, $\triangleleft(,)$ from rules Ty-term-Fill∗ are the 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).

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

$$\textsc{Ty-val-Hole}\quad \frac{}{\boxed{h} :_{1\nu} T \;{}^{\mathsf{v}}\!\vdash\; \boxed{h} : T}$$

$$\textsc{Ty-val-Dest}\quad \frac{}{\rightarrow\! h :_{1\nu} \lfloor_{n} T \rfloor \;{}^{\mathsf{v}}\!\vdash\; \rightarrow\! h : \lfloor_{n} T \rfloor}$$

$$\textsc{Ty-val-Unit}\quad \frac{}{\cdot \;{}^{\mathsf{v}}\!\vdash\; () : 1}$$

$$\textsc{Ty-val-Fun}\quad \frac{\Delta,\ x :_{m} T \vdash \mathrm{u} : U}{\Delta \;{}^{\mathsf{v}}\!\vdash\; {}^{\mathsf{v}}\lambda x \;{}_{m}\!\mapsto \mathrm{u} : T \;{}_{m}\!\rightarrow U}$$

$$\textsc{Ty-val-Left}\quad \frac{\Gamma \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}_1 : T_1}{\Gamma \;{}^{\mathsf{v}}\!\vdash\; \mathsf{Inl}\,\mathrm{v}_1 : T_1 \oplus T_2}$$

$$\textsc{Ty-val-Right}\quad \frac{\Gamma \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}_2 : T_2}{\Gamma \;{}^{\mathsf{v}}\!\vdash\; \mathsf{Inr}\,\mathrm{v}_2 : T_1 \oplus T_2}$$

$$\textsc{Ty-val-Prod}\quad \frac{\Gamma_1 \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}_1 : T_1 \qquad \Gamma_2 \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}_2 : T_2}{\Gamma_1 + \Gamma_2 \;{}^{\mathsf{v}}\!\vdash\; (\mathrm{v}_1\,,\,\mathrm{v}_2) : T_1 \otimes T_2}$$

$$\textsc{Ty-val-Exp}\quad \frac{\Gamma \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}' : T}{n\cdot\Gamma \;{}^{\mathsf{v}}\!\vdash\; \mathsf{E}_{n}\,\mathrm{v}' : !_{n} T}$$

$$\textsc{Ty-val-Ampar}\quad \frac{\begin{array}{c}\mathtt{LinOnly}\ \Delta_3 \qquad \mathtt{FinAgeOnly}\ \Delta_3 \\ 1\!\uparrow\!\cdot\Delta_1,\ \Delta_3 \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}_1 : T \\ \Delta_2,\ (\rightarrow^{-1}\Delta_3) \;{}^{\mathsf{v}}\!\vdash\; \mathrm{v}_2 : U\end{array}}{\Delta_1,\ \Delta_2 \;{}^{\mathsf{v}}\!\vdash\; {}_{\mathrm{hnames}(\Delta_3)}\langle \mathrm{v}_2 \;{}_{\wedge}\, \mathrm{v}_1 \rangle : U \ltimes T}$$

Fig. 9. Typing rules for values

5.3.2 *Typing of values* ${}^{\mathsf{v}}\!\vdash$.

Values v, presented as a subset of terms t, could be removed completely from the user syntax (given we promote alloc to a first-class keyword), and just used as a denotation for runtime data structures.

The typing of runtime values, given in Figure 9 is where hole and destination bindings appears. Values can have holes and destinations inside, but a value used as a term must not have any hole[4]. Also, variables cannot mention free variables; that makes it easier to prove substitution properties (as we will see in Section 6.3, we perform substitutions not only in terms, but also in evaluation contexts sometimes).

Hole bindings and destination bindings of the same hole name h are meant to annihilate each other in the typing context given they have a matching base type and mode. That way, the typing context of a term can stay constant during reduction:

- when destination-filling primitives are evaluated to build up data structures, they linearly consume a destination and write to a hole at the same time which makes both disappear, thus the typing context stays balanced;
- when a new hole is created, a matching destination is returned too, so the typing context stays balanced too.

However, the annihilation between a destination and a hole binding having the same name in the typing tree is only allowed to happen around an ampar, as it is the ampar connective that bind the name across the two sides (the names bound are actually stored in a set H on the ampar value ${}_{H}\langle \mathrm{v}_2 \;{}_{\wedge}\, \mathrm{v}_1 \rangle$). In fact, an ampar can be seen as a sort of lambda-abstraction, whose body (containing holes instead of variables) and input sink (destinations) are split across two sides, and magically interconnected through the ampar connective.

In most rules, we use a sum $\Gamma_1 + \Gamma_2$ for typing contexts (or the disjoint variant $\Gamma_1,\ \Gamma_2$). This sum doesn't not allow for annihilation of bindings with the same name; the operation is partial, and in particular it isn't defined if a same hole name is present in the operands in the two different forms (hole binding and destination binding). In particular, a pair $(\boxed{h}\,,\ \rightarrow\! h)$ is not well-typed. A single typing context Γ is not allowed either to contain both a hole binding and a destination binding for the same hole name.

[4]see Ty-term-Val: the value must type in a context Δ which means "destination only" by convention

Typing of ampars. As stated above, the core idea of Ty-val-Ampar is to act as a binding connective for hole and destinations.

We define a new operator $\rightarrow^{-1}$ to represent the matching hole bindings for a set of destination bindings. It is a partial, point-wise operation on typing bindings of a context where:

$$\rightarrow^{-1}(\rightarrow h :_{1\nu} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor) \quad = \quad \boxed{h} :_{\mathsf{n}} \mathsf{T}$$

Only an input context Δ made only of destination bindings, with mode 1ν, results in a valid output context (which is then only composed of hole bindings).

Equipped with this operator, we introduce the annihilation of bindings using Δ_3 to represent destinations on the right side v_1, and $\rightarrow^{-1}\Delta_3$ to represent the matching hole bindings on the left side v_2 (the structure under construction). Those bindings are only present in the premises of the rule but get removed in the conclusion. Those hole names are only local, bound to that ampar and don't affect the outside world nor can be referenced from the outside.

Both sides of the ampar may also contain stored destinations from other scopes, represented by $1\!\uparrow\!\cdot\Delta_1$ and Δ_2 in the respective typing contexts of v_1 and v_2. All holes introduced by this ampar have to be annihilated by matching destinations; following our naming convention, no hole binding can appear in $\Delta_1,\ \Delta_2$ in the conclusion.

Not sure whether we should explain why Δ_1 is offset by $1\!\uparrow$ in the premise but not in conclusion. The reason is "because it is needed" more than having a good intuition narrative behind it.

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

Other notable typing rules for values. Rules Ty-val-Hole and Ty-val-Dest indicates that a hole or destination must have mode 1ν in the typing context to be used (except when a destination is stored away, as we have seen).

revisit this if we allow weakening for dests

Rules for unit, left and right variants, and product are straightforward.

Rule Ty-val-Exp is rather classic too: we multiply the dependencies Γ of the value by the mode n of the exponential. The intuition is that if v uses a resource v' twice, then $\mathrm{E}_2\ \mathrm{v}$, that corresponds to two uses of v (in a system with such a mode), will use v' four times.

Rule Ty-val-Fun indicates that (value level) lambda abstractions cannot have holes inside. In other terms, a function value cannot be built piecemeal like other data structures, its whole body must be a complete term right from the beginning. It cannot contain free variables either, as the body of the function must type in context $\Delta,\ x :_{\mathsf{m}} \mathsf{T}$ where Δ is made only of destination bindings. One might wonder, how can we represent a curryfied 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 ${}^{\mathrm{v}}\lambda x \mapsto \textbf{from}'_{\ltimes}(\textbf{map alloc with } d \mapsto d \triangleleft (\lambda y \mapsto x \textbf{ concat } y))$, where the inner closure is not yet in value form, but pending to be built into a value. As the form $d \triangleleft (\lambda y \mapsto \mathrm{t})$ is part of term syntax, and not value syntax, we allow free variable captures in it.

6 EVALUATION CONTEXTS AND SEMANTICS

The semantics of λ_d are given using small-step reductions on a pair $\mathrm{C}[\mathrm{t}]$ of an evaluation context C (represented by a stack) determining a focusing path, and a term t under focus. Such a pair $\mathrm{C}[\mathrm{t}]$ is called a *command*, and represents a running program. We think that our semantics makes it easier to reason about types and linearity of a running program with holes than a store-based approach such as the one previewed in Section **??**.

6.1 Evaluation contexts forms

The grammar of evaluation contexts is given in Figure 10. An evaluation context C is the composition of an arbitrary number of focusing components $\mathrm{c}_1, \mathrm{c}_2 \ldots$. We chose to represent this composition

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

Fig. 10. Grammar for evaluation contexts

explicitly using a stack, instead of a meta-operation that would only let us access its final result. As a result, focusing and defocusing operations are made explicit in the semantics, resulting in a more verbose but simpler proof. It is also easier to imagine how to build a stack-based interpreter for such a language.

Focusing components are all directly derived from the term syntax, except for the "open ampar" focus component ${}^{\mathrm{op}}_{H}\langle \mathrm{v}_2 \,{}_{\wedge} \square \rangle$. This focus component indicates that an ampar is currently being **map**ped on, with its left-hand side v_2 (the structure being built) being attached to the "open ampar" focus component, while its right-hand side (containing destinations) is either in subsequent focus components, or in the term under focus.

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

6.2 Typing of evaluation contexts and commands

Evaluation contexts are typed in a context Δ that can only contains destination bindings. Δ represents the typing context available for the term that will be put in the box $\square$ of the evaluation context. In other terms, while the typing context of a term is a list of requirements so that it can be typed, the typing context of an evaluation context is the set of bindings that it makes available to the term under focus. As a result, while the typing of a term $\Theta \vdash \mathrm{t} : \mathsf{T}$ is additive (the typing requirements for a function application is the sum of the requirements for the function itself and for its argument), the typing of an evaluation context $\Delta \dashv \mathrm{C} : \mathsf{T} \rightarrowtail \mathsf{U}_0$ is subtractive : adding the focus component $\mathrm{t}'\,\square$ to the stack C will remove whatever is needed to type t' from the typing context provided by C. The whole typing rules for evaluation contexts C as well as commands $\mathrm{C}[\mathrm{t}]$ are presented in Figure 11.

An evaluation context has a pseudo-type $\mathsf{T} \rightarrowtail \mathsf{U}_0$, where T denotes the type of the focus (i.e. the type of the term that can be put in the box of the evaluation context) while U_0 denotes the type of the resulting command (when the box of the evaluation context is filled with a term).

Composing an evaluation context of pseudo-type $\mathsf{T} \rightarrowtail \mathsf{U}_0$ with a new focus component never affects the type U_0 of the future command ; only the type T of what can be put in the box is altered.

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

- the typing context $\mathrm{m} \cdot \Delta_1 + \Delta_2$ in the premise for C corresponds to $\mathrm{m} \cdot \Theta_1 + \Theta_2$ in the conclusion of Ty-term-PatP in Figure 7;

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

$$\textsc{Ty-ectxs-Id}\quad \frac{}{\cdot \dashv \square : \mathrm{U}_0 \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-App-Foc1}\quad \frac{\mathrm{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathrm{C} : \mathrm{U} \rightarrowtail \mathrm{U}_0 \qquad \Delta_2 \vdash \mathrm{t}' : \mathrm{T}\ {}_{\mathrm{m}}\!\!\to \mathrm{U}}{\Delta_1 \dashv \mathrm{C} \circ (\mathrm{t}'\ \square) : \mathrm{T} \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-App-Foc2}\quad \frac{\mathrm{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathrm{C} : \mathrm{U} \rightarrowtail \mathrm{U}_0 \qquad \Delta_1 \vdash \mathrm{v} : \mathrm{T}}{\Delta_2 \dashv \mathrm{C} \circ (\square\ \mathrm{v}) : (\mathrm{T}\ {}_{\mathrm{m}}\!\!\to \mathrm{U}) \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-PatU-Foc}\quad \frac{\Delta_1,\ \Delta_2 \dashv \mathrm{C} : \mathrm{U} \rightarrowtail \mathrm{U}_0 \qquad \Delta_2 \vdash \mathrm{u} : \mathrm{U}}{\Delta_1 \dashv \mathrm{C} \circ (\square \ ⨾\ \mathrm{u}) : 1 \rightarrowtail \mathrm{U}_0}$$

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

$$\textsc{Ty-ectxs-PatP-Foc}\quad \frac{\mathrm{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathrm{C} : \mathrm{U} \rightarrowtail \mathrm{U}_0 \qquad \Delta_2,\ x_1 :_{\mathrm{m}} \mathrm{T}_1,\ x_2 :_{\mathrm{m}} \mathrm{T}_2 \vdash \mathrm{u} : \mathrm{U}}{\Delta_1 \dashv \mathrm{C} \circ (\mathbf{case}_{\mathrm{m}}\ \square\ \mathbf{of}\ (x_1\,,\ x_2) \mapsto \mathrm{u}) : (\mathrm{T}_1 \otimes \mathrm{T}_2) \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-PatE-Foc}\quad \frac{\mathrm{m}\cdot\Delta_1,\ \Delta_2 \dashv \mathrm{C} : \mathrm{U} \rightarrowtail \mathrm{U}_0 \qquad \Delta_2,\ x :_{\mathrm{m}\cdot\mathrm{m}'} \mathrm{T} \vdash \mathrm{u} : \mathrm{U}}{\Delta_1 \dashv \mathrm{C} \circ (\mathbf{case}_{\mathrm{m}}\ \square\ \mathbf{of}\ \mathsf{E}_{\mathrm{m}'}\ x \mapsto \mathrm{u}) : !_{\mathrm{m}'}\mathrm{T} \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-Map-Foc}\quad \frac{\Delta_1,\ \Delta_2 \dashv \mathrm{C} : \mathrm{U} \ltimes \mathrm{T}' \rightarrowtail \mathrm{U}_0 \qquad 1\!\uparrow\cdot\Delta_2,\ x :_{1\nu} \mathrm{T} \vdash \mathrm{t}' : \mathrm{T}'}{\Delta_1 \dashv \mathrm{C} \circ (\mathbf{map}\ \square\ \mathbf{of}\ x \mapsto \mathrm{t}') : (\mathrm{U} \ltimes \mathrm{T}) \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-ToA-Foc}\quad \frac{\Delta \dashv \mathrm{C} : (\mathrm{U} \ltimes 1) \rightarrowtail \mathrm{U}_0}{\Delta \dashv \mathrm{C} \circ (\mathbf{to}_{\ltimes}\ \square) : \mathrm{U} \rightarrowtail \mathrm{U}_0}$$

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

$$\textsc{Ty-ectxs-FillU-Foc}\quad \frac{\Delta \dashv \mathrm{C} : 1 \rightarrowtail \mathrm{U}_0}{\Delta \dashv \mathrm{C} \circ (\square \triangleleft ()) : \lfloor_{\mathrm{n}} 1 \rfloor \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-FillL-Foc}\quad \frac{\Delta \dashv \mathrm{C} : \lfloor_{\mathrm{n}} \mathrm{T}_1 \rfloor \rightarrowtail \mathrm{U}_0}{\Delta \dashv \mathrm{C} \circ (\square \triangleleft \mathsf{Inl}) : \lfloor_{\mathrm{n}} \mathrm{T}_1 \oplus \mathrm{T}_2 \rfloor \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-FillR-Foc}\quad \frac{\Delta \dashv \mathrm{C} : \lfloor_{\mathrm{n}} \mathrm{T}_2 \rfloor \rightarrowtail \mathrm{U}_0}{\Delta \dashv \mathrm{C} \circ (\square \triangleleft \mathsf{Inr}) : \lfloor_{\mathrm{n}} \mathrm{T}_1 \oplus \mathrm{T}_2 \rfloor \rightarrowtail \mathrm{U}_0}$$

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

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

$$\textsc{Ty-ectxs-FillF-Foc}\quad \frac{\Delta_1,\ (1\!\uparrow\cdot\mathrm{n})\cdot\Delta_2 \dashv \mathrm{C} : 1 \rightarrowtail \mathrm{U}_0 \qquad \Delta_2,\ x :_{\mathrm{m}} \mathrm{T} \vdash \mathrm{u} : \mathrm{U}}{\Delta_1 \dashv \mathrm{C} \circ (\square \triangleleft (\lambda x\ {}_{\mathrm{m}}\!\!\mapsto \mathrm{u})) : \lfloor_{\mathrm{n}} \mathrm{T}\ {}_{\mathrm{m}}\!\!\to \mathrm{U} \rfloor \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-FillComp-Foc1}\quad \frac{\Delta_1,\ (1\!\uparrow\cdot\mathrm{n})\cdot\Delta_2 \dashv \mathrm{C} : \mathrm{T} \rightarrowtail \mathrm{U}_0 \qquad \Delta_2 \vdash \mathrm{t}' : \mathrm{U} \ltimes \mathrm{T}}{\Delta_1 \dashv \mathrm{C} \circ (\square \triangleleft\!\bullet\, \mathrm{t}') : \lfloor_{\mathrm{n}} \mathrm{U} \rfloor \rightarrowtail \mathrm{U}_0}$$

$$\textsc{Ty-ectxs-FillComp-Foc2}\quad \frac{\Delta_1,\ (1\!\uparrow\cdot\mathrm{n})\cdot\Delta_2 \dashv \mathrm{C} : \mathrm{T} \rightarrowtail \mathrm{U}_0 \qquad \Delta_1 \vdash \mathrm{v} : \lfloor_{\mathrm{n}} \mathrm{U} \rfloor}{\Delta_2 \dashv \mathrm{C} \circ (\mathrm{v} \triangleleft\!\bullet\, \square) : \mathrm{U} \ltimes \mathrm{T} \rightarrowtail \mathrm{U}_0}$$

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

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

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

Fig. 11. Typing rules for evaluation contexts and commands

- the typing context $\Delta_2,\ x_1 :_{\mathrm{m}} \mathrm{T}_1,\ x_2 :_{\mathrm{m}} \mathrm{T}_2$ in the premise for term u corresponds to the typing context $\Theta_2,\ x_1 :_{\mathrm{m}} \mathrm{T}_1,\ x_2 :_{\mathrm{m}} \mathrm{T}_2$ for the same term in Ty-term-PatP;
- the typing context Δ_1 in the conclusion for $\mathrm{C} \circ (\mathbf{case}_{\mathrm{m}}\ \square\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathrm{u})$ corresponds to the typing context Θ_1 in the premise for t in Ty-term-PatP (the term t is located where the box $\square$ is in Ty-ectxs-OpenAmpar-Foc).

In a way, the typing rule for an evaluation context is a "rotation" of the typing rule for the associated term, where the typing contexts of one premise and the conclusion are swapped, an the typing context of the other potential premise is kept unchanged (with the added difference that free variables cannot appear in typing contexts of evaluation contexts, so any Θ becomes a Δ).

As we see at the bottom of the figure, a command $\mathrm{C}[\mathrm{t}]$ (i.e. a pair of an evaluation context and a term) is well typed when the evaluation context C provides a typing context Δ that is exactly one in which t is well typed. We can always embed a well-typed, closed term $\bullet \vdash \mathrm{t} : \mathrm{T}$ as a well-typed command using the identity evaluation context: $\mathrm{t} \simeq \square[\mathrm{t}]$ and we thus have $\vdash \square[\mathrm{t}] : \mathrm{T}$ where $\Delta = \bullet$ (the empty context).

6.3 Small-step semantics

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

- focus rules, where we remove a layer from term t (which cannot be a value) and push a corresponding focus component on the stack C;
- unfocus rules, where t is a value and thus we pop a focus component from the stack C and transform it back to a 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 is the whole set of rules for PatP:

$$\textsc{Sem-PatP-Foc}\ \frac{\mathtt{NotVal}\ \mathrm{t}}{\mathrm{C}[\mathbf{case}_{\mathrm{m}}\ \mathrm{t}\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathrm{u}] \longrightarrow (\mathrm{C} \circ (\mathbf{case}_{\mathrm{m}}\ \square\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathrm{u}))[\mathrm{t}]}$$

$$\textsc{Sem-PatP-Unfoc}\ \frac{}{(\mathrm{C} \circ (\mathbf{case}_{\mathrm{m}}\ \square\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathrm{u}))[\mathrm{v}] \longrightarrow \mathrm{C}[\mathbf{case}_{\mathrm{m}}\ \mathrm{v}\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathrm{u}]}$$

$$\textsc{Sem-PatP-Red}\ \frac{}{\mathrm{C}[\mathbf{case}_{\mathrm{m}}\ (\mathrm{v}_1\,,\, \mathrm{v}_2)\ \mathbf{of}\ (x_1\,,\, x_2) \mapsto \mathrm{u}] \longrightarrow \mathrm{C}[\mathrm{u}[x_1 \coloneqq \mathrm{v}_1][x_2 \coloneqq \mathrm{v}_2]]}$$

Rules are triggered in a purely deterministic fashion; once a subterm is a value, it cannot be focused again. As focusing and defocusing rules are entirely mechanical (they are just a matter of pushing and popping a focus component on the stack), we only present the set of reduction rules for the system in Figure 12.TODO: Add full rules in annex ?

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 substitution $\mathrm{C}[h \coloneqq_H \mathrm{v}]$ on the evaluation context C that corresponds to a memory update of a hole $\boxed{h}$. Sem-FillU-Red and Sem-FillF-Red do not create any new hole; they only write a value to an existing one. On the other hand, rules Sem-FillL-Red, Sem-FillR-Red, Sem-FillE-Red and Sem-FillP-Red 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.

Obtaining a fresh name is represented by the statement $h' = \max(\mathrm{hnames}(\mathrm{C}) \cup \{h\})+1$ in the premises of these rules. One invariant of the system is that an ampar must have fresh names to be opened, so we always rename local hole names bound by an ampar to fresh names just when that

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

$$\textsc{Sem-App-Red}\quad \frac{}{\mathrm{C}[({}^{\mathsf{v}}\lambda x\ {}_{m}\!\mapsto \mathrm{u})\ \mathrm{v}] \longrightarrow \mathrm{C}[\mathrm{u}[x \coloneqq \mathrm{v}]]}$$

$$\textsc{Sem-PatU-Red}\quad \frac{}{\mathrm{C}[()\ \fatsemi\ \mathrm{u}] \longrightarrow \mathrm{C}[\mathrm{u}]}$$

$$\textsc{Sem-PatL-Red}\quad \frac{}{\mathrm{C}[\mathsf{case}_{m}\ (\mathsf{Inl}\ \mathrm{v}_1)\ \mathsf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathrm{u}_1\,,\ \mathsf{Inr}\ x_2 \mapsto \mathrm{u}_2\}] \longrightarrow \mathrm{C}[\mathrm{u}_1[x_1 \coloneqq \mathrm{v}_1]]}$$

$$\textsc{Sem-PatR-Red}\quad \frac{}{\mathrm{C}[\mathsf{case}_{m}\ (\mathsf{Inr}\ \mathrm{v}_2)\ \mathsf{of}\ \{\mathsf{Inl}\ x_1 \mapsto \mathrm{u}_1\,,\ \mathsf{Inr}\ x_2 \mapsto \mathrm{u}_2\}] \longrightarrow \mathrm{C}[\mathrm{u}_2[x_2 \coloneqq \mathrm{v}_2]]}$$

$$\textsc{Sem-PatP-Red}\quad \frac{}{\mathrm{C}[\mathsf{case}_{m}\ (\mathrm{v}_1\,,\mathrm{v}_2)\ \mathsf{of}\ (x_1\,,x_2) \mapsto \mathrm{u}] \longrightarrow \mathrm{C}[\mathrm{u}[x_1 \coloneqq \mathrm{v}_1][x_2 \coloneqq \mathrm{v}_2]]}$$

$$\textsc{Sem-PatE-Red}\quad \frac{}{\mathrm{C}[\mathsf{case}_{m}\ \mathsf{E}_{n}\ \mathrm{v}'\ \mathsf{of}\ \mathsf{E}_{n}\ x \mapsto \mathrm{u}] \longrightarrow \mathrm{C}[\mathrm{u}[x \coloneqq \mathrm{v}']]}$$

$$\textsc{Sem-ToA-Red}\quad \frac{}{\mathrm{C}[\mathsf{to}_{\ltimes}\ \mathrm{v}_2] \longrightarrow \mathrm{C}[{}_{\{\}}\langle \mathrm{v}_2\ {}_{\wedge}\,()\rangle]}$$

$$\textsc{Sem-FromA-Red}\quad \frac{}{\mathrm{C}[\mathsf{from}_{\ltimes}\ {}_{\{\}}\langle \mathrm{v}_2\ {}_{\wedge}\,\mathsf{E}_{1\infty}\ \mathrm{v}_1\rangle] \longrightarrow \mathrm{C}[(\mathrm{v}_2\,,\mathsf{E}_{1\infty}\ \mathrm{v}_1)]}$$

$$\textsc{Sem-FillU-Red}\quad \frac{}{\mathrm{C}[\rightarrow h \triangleleft ()] \longrightarrow \mathrm{C}[h \coloneqq_{\{\}}\ ()][()]}$$

$$\textsc{Sem-FillF-Red}\quad \frac{}{\mathrm{C}[\rightarrow h \triangleleft (\lambda x\ {}_{m}\!\mapsto \mathrm{u})] \longrightarrow \mathrm{C}[h \coloneqq_{\{\}}\ {}^{\mathsf{v}}\lambda x\ {}_{m}\!\mapsto \mathrm{u}][()]}$$

$$\textsc{Sem-FillL-Red}\quad \frac{h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}) \cup \{h\})+1}{\mathrm{C}[\rightarrow h \triangleleft \mathsf{Inl}] \longrightarrow \mathrm{C}[h \coloneqq_{\{h'+1\}}\ \mathsf{Inl}\ \boxed{h'+1}][\rightarrow h'+1]}$$

$$\textsc{Sem-FillR-Red}\quad \frac{h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}) \cup \{h\})+1}{\mathrm{C}[\rightarrow h \triangleleft \mathsf{Inr}] \longrightarrow \mathrm{C}[h \coloneqq_{\{h'+1\}}\ \mathsf{Inr}\ \boxed{h'+1}][\rightarrow h'+1]}$$

$$\textsc{Sem-FillE-Red}\quad \frac{h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}) \cup \{h\})+1}{\mathrm{C}[\rightarrow h \triangleleft \mathsf{E}_{m}] \longrightarrow \mathrm{C}[h \coloneqq_{\{h'+1\}}\ \mathsf{E}_{m}\ \boxed{h'+1}][\rightarrow h'+1]}$$

$$\textsc{Sem-FillP-Red}\quad \frac{h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}) \cup \{h\})+1}{\mathrm{C}[\rightarrow h \triangleleft (,)] \longrightarrow \mathrm{C}[h \coloneqq_{\{h'+1,h'+2\}}\ (\boxed{h'+1}\,,\boxed{h'+2})][(\rightarrow h'+1\,,\rightarrow h'+2)]}$$

$$\textsc{Sem-FillComp-Red}\quad \frac{h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}) \cup \{h\})+1}{\mathrm{C}[\rightarrow h \triangleleft\!\bullet\ {}_{H}\langle \mathrm{v}_2\ {}_{\wedge}\,\mathrm{v}_1\rangle] \longrightarrow \mathrm{C}[h \coloneqq_{(H \pm h')}\ \mathrm{v}_2[H \pm h']][\mathrm{v}_1[H \pm h']]}$$

$$\textsc{Sem-Map-Red-OpenAmpar-Foc}\quad \frac{h' = \mathsf{max}(\mathsf{hnames}(\mathrm{C}))+1}{\mathrm{C}[\mathsf{map}\ {}_{H}\langle \mathrm{v}_2\ {}_{\wedge}\,\mathrm{v}_1\rangle\ \mathsf{with}\ x \mapsto \mathrm{t}'] \longrightarrow (\mathrm{C} \circ ({}^{\mathrm{op}}_{H \pm h'}\langle \mathrm{v}_2[H \pm h']\ {}_{\wedge}\,\square\rangle))[\mathrm{t}'[x \coloneqq \mathrm{v}_1[H \pm h']]]}$$

$$\textsc{Sem-OpenAmpar-Unfoc}\quad \frac{}{(\mathrm{C} \circ^{\mathrm{op}}_{H} \langle \mathrm{v}_2\ {}_{\wedge}\,\square\rangle)[\mathrm{v}_1] \longrightarrow \mathrm{C}[{}_{H}\langle \mathrm{v}_2\ {}_{\wedge}\,\mathrm{v}_1\rangle]}$$

Fig. 12. Small-step semantics

ampar is **map**ped on, as these local names — represented by the set of hole names H that the ampar carries — could otherwise shadow already existing names in evaluation context C. This invariant is materialized by premise $\mathbf{hnames}(\mathrm{C})$ ## $\mathbf{hnames}(\Delta_3)$ in rule Ty-ectxs-OpenAmpar-Foc for the open ampar focus component that is created during reduction of a **map**.

We use hole name shifting as a strategy to obtain fresh names. Shifting all hole names in a set H by a given offset h' is denoted $H\pm h'$. We extend this notation to define a conditional shift operation $[H\pm h']$ which shifts each hole name appearing in the operand to the left of the brackets by h' if this hole name is also member of H. This conditional shift can be used on a single hole name, a value, or a typing context.

In rule Sem-FillComp-Red, we write the left-hand side 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 some focus fragment ${}^{\mathrm{op}}_{H'}\langle \mathrm{v}_2' \,{}_\wedge\, \square \rangle$ in the evaluation context C. That fragment is not mentioned explicitly in the rule, as the destination $\rightarrow h$ is enough to target it. This results in the composition of two structures with holes v_2' and v_2 through filling of $\rightarrow h$. Because we split open the ampar ${}_H\langle \mathrm{v}_2 \,{}_\wedge\, \mathrm{v}_1 \rangle$ (its left-hand side gets written to a hole, while its right hand side is returned), we need to rename any hole name that it contains to a fresh one, as we do when an ampar is opened in the **map** rule. The renaming is carried out by the conditional shift $\mathrm{v}_2[H\pm h']$ and $\mathrm{v}_1[H\pm h']$ (only hole names local to the ampar, represented by the set H, gets renamed).

Last but not least, rules Sem-Map-Red-OpenAmpar-Foc and Sem-OpenAmpar-Unfoc dictates how and when a closed ampar (a term) is converted to an open ampar (a focusing fragment) and vice-versa. With Sem-Map-Red-OpenAmpar-Foc, the local hole names of the ampar gets renamed to fresh ones, and the left-hand side gets attached to the focusing fragment ${}^{\mathrm{op}}_{H\pm h'}\langle \mathrm{v}_2[H\pm h'] \,{}_\wedge\, \square \rangle$ while the right-hand side (containing destinations) is substituted in the body of the **map** statement (which becomes the new term under focus). This effectively allows the right-hand side of an ampar to be a term instead of a value for a limited time.

The rule Sem-OpenAmpar-Unfoc triggers when the body of a **map** statement has reduced to a value. In that case, we can close the ampar, by popping the focus fragment from the stack C and merging back with v_2 to reform a closed ampar.

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* $\forall \mathrm{v}, \mathrm{C}[\mathrm{t}] \neq \square[\mathrm{v}]$ *then* $\exists \mathrm{C}', \mathrm{t}'.\ \mathrm{C}[\mathrm{t}] \longrightarrow \mathrm{C}'[\mathrm{t}']$.

A command of the form $\square[\mathrm{v}]$ cannot be reduced further, as it only contains a fully determined value, and no pending computation. This it is the expected stopping point of the reduction, and any well-typed command is supposed to reach such a form at some point.

7 FORMAL PROOF OF TYPE SAFETY

We've proved type preservation and progress theorems with the Coq proof assistant. At time of writing, we have assumed, rather than proved, the substitution lemmas. The choice of turning to a proof assistant was a pragmatic choice: the context handling in λ_d can be quite finicky, and it was hard, without computer assistance, to make sure that we hadn't made mistakes in our proofs. The version of λ_d that we've proved is written in Ott, 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; some visual simplification is applied by a script to produce the version in the article.

We probably want to make sure that the statement of these two lemmas is stated in the type system section

a citation maybe

Most of the proof was done by an author with little prior experience with Coq. This goes to show that Coq is reasonably approachable even for non-trivial development. The proof is about 6000 lines long, and contains nearly 350 lemmas. Many of the cases of the type preservation and progress lemmas are similar, to handle such repetitive cases using of a large-language-model based autocompletion system has been quite effective.

Binders are the biggest problem. We've largely manage to make the proof to be only about closed terms, to avoid any complication with binders. This worked up until the substitution lemmas, which is the reason why we haven't proved them in Coq yet (that and the fact that it's much easier to be confident in our pen-and-paper proofs for those). There are backends to generate locally nameless citation representations from Ott definitions; we haven't tried them yet, but the unusual binding nature of ampars may be too much for them to handle.

The proofs aren't very elegant. For instance, we don't have any abstract formalization of semirings: since our semirings are finite 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 bunch 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.
- We make the mode semiring total by adding an invalid mode. This prevents us from having to deal with partiality at all. The cost is that contexts can contain binders with invalid mode. Proofs are written so as to rule out this case.

The inference rules produced by Ott aren't conducive to using setoid equality. This turned out to be a problem with our type for finite function:

```
Record T A B := {
    underlying :> forall x:A, option (B x);
    supported : exists l : list A, Support l underlying;
  }.
```

where `Support l f` means that `l` contains the domain of `f`. To make the equality of finite function be strict equality `eq`, we assumed functional extensionality and proof irrelevance. In some circumstances, we've also needed to list the finite functions' domains. But in the definition, the domain is sealed behind a proposition, so we also assumed classical logic as well as indefinite description

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

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

I don't think we need to write this: methodology: assume a lot of lemmas, prove main theorem, prove assumptions, some wrong, fix. A number of wrong lemma initially assumed, but replacing them by correct variant was always easy to fix in proofs.

8 IMPLEMENTATION OF λ_d USING IN-PLACE MEMORY MUTATIONS

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

First, λ_d misses a form of recursion, but we believe that adding equirecursive types and a fix-point operator wouldn't compromise the safety of the system.

Secondly, ampars are not managed linearly in λ_d ; only destinations are. That is to say that an ampar can be wrapped in an exponential, e.g. $\mathsf{E}_{\omega\nu}\,{}_{\{h\}}\langle \mathsf{Inr}\,(\mathsf{Inl}\,()\,,\ \boxed{h})\ {}_{\wedge}\!\rightarrow h\rangle$ (representing a non-linear difference list $\mathsf{E}_{\omega\nu}\,(0 :: \square)$), and then used twice, each time in a different way:

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

It may seem counter-intuitive at first, but this program is valid and safe in λ_d . Thanks to the renaming discipline we detailed in Section 6.3, every time an ampar is **map**ped over, its hole names are renamed to fresh ones. So when we call **append** to build x_1 (which is implemented in terms of **map**), we sort of allocate a new copy of the ampar before mutating it, effectively achieving a copy-on-write memory scheme. Thus it is safe to operate on x again to build x_2.

In the introduction of the article, we announced a safe framework for in-place memory mutation, so we will uphold this promise now. The key to go from a copy-on-write scheme to an in-place mutation scheme is to force ampars to be linearly managed too. For that we introduce a new type Token, together with primitives **dup** and **drop** (remember that unqualified arrows have mode 1ν, so are linear):

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

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

We use the Token type as an intrinsic source of linearity that infects the ampar returned by $\mathsf{alloc}_{\mathsf{ip}}$. Such a token can be duplicated using **dup**, but as soon as it is used to create an ampar, that ampar cannot be duplicated itself. In the system featuring the Token type and $\mathsf{alloc}_{\mathsf{ip}}$, "closed" programs now typecheck in the non-empty context $\{tok_0 :_{1\infty} \mathsf{Token}\}$ containing a token variable that the user can **dup**licate and **drop** freely to give birth to an arbitrary number of ampars, that will then have to be managed linearly.

Having closed programs to typecheck in non-empty context $\{tok_0 :_{1\infty} \mathsf{Token}\}$ is very similar to having a primitive function **withToken** $:$ $(\mathsf{Token}\ {}_{1\infty}\!\!\rightarrow\ !_{\omega\infty}\mathsf{T}) \rightarrow\ !_{\omega\infty}\mathsf{T}$ as it is done in [Bagrel 2024].

In such an extension, as ampars are managed linearly, we can change the allocation and renaming mechanisms:

- the hole name for a new ampar can be 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 "fillComp" ($\vartriangleleft\!\bullet$) no longer require any renaming: we have the guarantee that the names are globally fresh, and thus we can do in-place memory updates.

We decided to omit the linearity aspect of ampars in λ_d as it clearly obfuscate the presentation of the system without adding much to the understanding of the latter. We believe that the system is still sound with this linearity aspect, and articles such as [Spiwack et al. 2022] gives a pretty clear view on how to implement the linearity requirement for ampars in practice without too much noise for the user.

9 RELATED WORK

Subsection for each work seems a little heavy-weight. On the other hand the italic paragraphs seem

9.1 Destination-passing style for efficient memory management

In [Shaikhha et al. 2017], the authors present a destination-based intermediate language for a functional array programming language. They develop a system of destination-specific optimizations and boast near-C performance.

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

The main difference between their work and ours is that their language is solely an intermediate language: it would be unsound to program in it manually. We, on the other hand, are proposing a type system to make it sound for the programmer to program directly with destinations.

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

9.2 Tail modulo constructor

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

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

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

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

This has been available in OCaml since version 4.14. This is the one example we know of of destinations built in a production-grade compiler. Our λ_d makes it possible to express the result tail-modulo-constructor in a typed language. It can be used to write programs directly in that style, or it could serve as a typed target language for and automatic transformation. On the flip-side, tail modulo constructor is too weak to handle our difference lists or breadth-first traversal examples.

TODO: Mention Tail modulo context

9.3 A functional representation of data structures with a hole

The idea of using linear types to safely represent structures with holes 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, 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 [Minamide 1998] can implement our examples with difference lists and queues from Section 2.2, but it can't do our breadth-first traversal example from Section 4, since storing destinations in a data structure is the essential ingredient of this example.

This ability to store destination does come at a cost though: the system needs this additional notion of ages to ensure that destinations are use soundly. On the other hand, our system is strictly more general, in that the system from [Minamide 1998] can be embedded in λ_d , and if one stays in this fragment, we're never confronted with ages. Ages only show up when writing programs that go beyond Minamide's system.

9.4 Destination-passing style programming: a Haskell implementation

In [Bagrel 2024], the author proposes a system much like ours: it has a par-like construct (that they call Incomplete), where only the right-hand side can be modified, and a destination type. The main difference is that in their system, $d \blacktriangleleft \mathrm{t}$ requires t to be unrestricted, while in λ_d , t can be linear.

More of these grey ts I don't know why that is.

The consequence is that in [Bagrel 2024], destinations can be stored in data structures but not in data structures with holes. In order to do a breadth-first search algorithm like in Section 4, they can't use improved queues like we do, they have to use regular functional queues.

However, unlike λ_d , [Bagrel 2024] is implemented in Haskell, which features linear types. Our λ_d , with the age modes, needs more than what Haskell provides. Our system subsumes theirs, however, ages will appear in the typing rules for that fragment.

9.5 Semi-axiomatic sequent calculus

In, 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 $\lfloor \mathrm{T} \rfloor$ type, their dualization connective is involutive.

Add citation

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

10 CONCLUSION AND FUTURE WORK

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

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

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

REFERENCES

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

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

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

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

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

Daan Leijen and Anton Lorenzen. 2023. Tail Recursion Modulo Context: An Equational Approach. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1152–1181. https://doi.org/10.1145/3571233

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

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