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 destination calculus, 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 destination calculus 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, 32 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 the idea of explicitly receiving a location where to return a value to (we say that the destination is filled when the supplied location is written to). Instead of a function with signature $T \to U$, in λ_d you would have $T \to \lfloor U \rfloor \to 1$, where $\lfloor U \rfloor$ is read "destination of type U". For instance, here is a destination-passing version of the identity function:

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

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 whole value in a single step. Destinations can be filled piecemeal.

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

In this example, we're building a value of type $T \oplus U$ 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 □, write the address of that block to the location that d points to, and return a destination pointing to the uninitialized argument of Inl.

Notice that we are constructing the term from the outermost constructor inward: we've built a value of the form Inl □ but we have yet to describe the constructor's payload (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 9). 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:

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

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.

Values 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 $\lfloor T \rfloor$ while it is used to build a structure of type $T \oplus U$. To represents this, λ_d uses a type $S \ltimes \lfloor T \rfloor$ for a structure of type S missing a value of type T to be complete (we

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

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

Introduce the terminology "hollow constructor" here?

For some reason this v is grey. → Thomas : that's the case for non-terminals in the grammar

Make sure that we clarify early on that when we say "structure" we mean "linked data structure"

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 form $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ is read "S ampar destination of 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 5.1. For now, it's sufficient to observe that $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ is akin to a $\mathsf{S} \parr \mathsf{T}^\perp$ in linear logic, indeed you can think of $\mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor$ as a (linear) function from T to S. That values 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].

Somewhere around this point it would be good to explain the difference between holes and destinations. → Thomas: see DIFF-HOLE-DEST

Destinations always exist within the context of a structure with holes. A hole of type T inside S represents the fact that S contains uninitialized memory that will have to hold a T for S to be readable without errors; it only denotes the absence of a value and thus cannot be manipulated directly. A destination $\lfloor \mathsf{T} \rfloor$, on the other hand, is a first-class value that witnesses the presence of a hole of type T inside S and can be used to complete the structure. To access the destinations, λ_d provides a **map** construction, which lets us apply a function to the right-hand side of an ampar:

DIFF-HOLE-DEST

$$\textbf{fillWithInl'} : \mathsf{S} \ltimes \lfloor \mathsf{T} \oplus \mathsf{U} \rfloor \to \mathsf{S} \ltimes \lfloor \mathsf{T} \rfloor \qquad \textbf{fillWithAPair'} : \mathsf{S} \ltimes \lfloor \mathsf{T} \otimes \mathsf{U} \rfloor \to \mathsf{S} \ltimes (\lfloor \mathsf{T} \rfloor \otimes \lfloor \mathsf{U} \rfloor)$$
$$\textbf{fillWithInl'}\ x \triangleq x \triangleright \textbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \qquad \textbf{fillWithAPair'}\ x \triangleq x \triangleright \textbf{map}\ d \mapsto d \triangleleft \mathsf{Inl}$$

It may be more readable to have fillWithInl' and fillWithAPair' side by side. Maybe they're too wide though

To tie this up, we need a way to introduce and to eliminate values with holes. Values with holes are introduced with **alloc** which creates a value $\mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor$, which is simply a hole expecting a value of type T. Values with holes are eliminated with[1] $\textbf{from}'_\ltimes : \mathsf{S} \ltimes 1 \to \mathsf{S}$: if all the destinations have been filled, then a value with holes is really just a normal, complete value.

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

$$\mathsf{Inl} : \mathsf{T} \to \mathsf{T} \oplus \mathsf{U}$$
$$\mathsf{Inl}\ x \triangleq \textbf{from}'_\ltimes\ (\textbf{alloc} \triangleright \textbf{map}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft x)$$

$$(,) : \mathsf{T} \to \mathsf{U} \to \mathsf{T} \otimes \mathsf{U}$$
$$(x, y) \triangleq \textbf{from}'_\ltimes\ (\textbf{alloc} \triangleright \textbf{map}\ d \mapsto (d \triangleleft (,)) \triangleright \textbf{case}\ (d_1, d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y)$$

Wording slightly awkward. Revisit at some point. Maybe point out that it makes destinations strictly more expressive than traditional constructor. Maybe discuss as Thomas did at some point that this means that the order in which a structure is built is actually arbitrary, not just outside-in.

Purity. At this point, the reader may be forgiven for feeling distressed at all the talk of filling by mutation and uninitialized memory. How is it consistent with our claim to be building a pure language?

Indeed, unrestricted use of destination can lead to rather catastrophic results. The simplest such issue happens when we forget to fill a destination:

$$\textbf{forget} : \mathsf{T}$$
$$\textbf{forget} \triangleq \textbf{from}'_\ltimes\ (\textbf{alloc} \triangleright \textbf{map}\ d \mapsto ())$$

Here, **forget** claims to be a value of type T, but it's really just a hole, just uninitialized memory. So any attempt to use **forget** will read uninitialized memory.

A similar situation can happen if we reuse a destination.

$$\textbf{overwrite} : \mathsf{T} \to \mathsf{T} \oplus \mathsf{U}$$
$$\textbf{overwrite}\ x \triangleq \textbf{from}'_\ltimes\ (\textbf{alloc} \triangleright \textbf{map}\ d \mapsto \textbf{let}\ d' \coloneqq d \triangleleft \mathsf{Inl}\ \textbf{in}\ d \triangleleft \mathsf{Inr} \fatsemi d' \blacktriangleleft x)$$

Thomas: I do not really like this paragraph. I would like this paragraph to be the other way around: we want a pure model, in particular a model that doesn't break observable immutability. So write-(exactly)-once memory model

These programs are rejected by λ_d , using a linear type system, ensuring that λ_d is a pure and safe language. This justifies the claim that we can think of destinations as write-once memory locations.

[1] As the name suggest, there is a more general elimination $\textbf{from}_\ltimes$. It will be discussed in Section 6.

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

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

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

Fig. 1. List implementation in equirecursive destination calculus

Linearity of destination isn't the only measure that we need to take in order to ensure that uninitialized memory is never read. Indeed consider a value of type $\mathsf{v} : \mathsf{S} \ltimes \mathsf{T}$. In memory v is a value of type S except that it has some uninitialized memory. So we have really careful when we read anything in v. In λ_d we take the bluntest restriction: we simply do not give any way to read v at all.

This is quite incorrect, v is rather a pair, and only its left side cannot be read. At least, if we stick to a general T on the right hand side, it would be too much of a lie to say that there isn't anything else than a S in memory

2.2 Functional queues, with destinations

If we suppose equirecursive types and a fixed-point operator, then λ_d becomes expressive enough to build any usual data structure.

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.

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

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

I wanted to have an example here, but I couldn't find a natural way to write something bad since we truly don't have a tool to write them.

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.

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.

In destination calculus, 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. This is exactly what an ampar is designed to allow: thanks to an ampar, we can handle incomplete structures safely, with no need to complete them immediately. The incomplete list is represented by the left side of the ampar, and the destination is represented by its right side. Creating an empty difference list is exactly what the **alloc** primitive already does when specialized to type $\mathsf{List}\ \mathsf{T}$: it returns an incomplete list with no items in it, and a destination pointing to that cell so that the

$$\mathsf{DList}\ T \triangleq (\mathsf{List}\ T) \ltimes \lfloor \mathsf{List}\ T \rfloor$$

$$\textbf{append} : \mathsf{DList}\ T \to T \to \mathsf{DList}\ T$$
$$ys\ \textbf{append}\ y \triangleq ys \triangleright \textbf{map}\ dys \mapsto (dys \triangleleft (::)) \triangleright \textbf{case}\ (dy, dys') \mapsto dy \blacktriangleleft y \fatsemi dys'$$

$$\textbf{concat} : \mathsf{DList}\ T \to \mathsf{DList}\ T \to \mathsf{DList}\ T$$
$$ys\ \textbf{concat}\ ys' \triangleq ys \triangleright \textbf{map}\ d \mapsto d \triangleleft\!\bullet\ ys'$$

$$\textbf{to}_{\mathsf{List}} : \mathsf{DList}\ T \to \mathsf{List}\ T$$
$$\textbf{to}_{\mathsf{List}}\ ys \triangleq \textbf{from}'_{\ltimes}\ (ys \triangleright \textbf{map}\ d \mapsto d \triangleleft [])$$

Fig. 2. Difference list implementation in equirecursive destination calculus

list can be built later through destination-filling primitives, together in an ampar: $\mathsf{List}\ T \ltimes \lfloor \mathsf{List}\ T \rfloor$. Type definition and operators for difference lists in destination calculus are presented in Figure 2.

The **append** simply appends an element at the end of the list. It uses "fillCons" to link a new hollow "cons" cell at the end of the list, and then handles the two associated destinations dy and dt. 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.

The **concat** operator concatenates two difference lists by writing the head of the second one to the hole left at the end of the first one. This is done using the "fillComp" primitive $\triangleleft\!\bullet : \lfloor_n U_1 \rfloor \to U_1 \ltimes U_2 \overset{1\uparrow \cdot n}{\longrightarrow} U_2$[2]. It takes a destination on its left-hand side, and an ampar on its right-hand side. The left side of the ampar (type U_1) is fed to the destination (so the incomplete structure is written to a larger incomplete structure from which the destination originated from), and the right side of the ampar (type U_2) is returned.

Here the left side of the second ampar is the second incomplete list, which is pasted at the end of the first incomplete list, consuming the destination of the first difference list in the process. Then the right side of the second ampar, that is to say the destination to the yet-unspecified tail of the second difference list, is returned, and stored back in the resulting ampar (thus serves as the new destination to the tail of the the resulting difference list).

Finally, the $\textbf{to}_{\mathsf{List}}$ 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 reversed 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 (just pattern-match on the first cons cell of the first list, $O(1)$), and enqueuing a new element is efficient too (just add a new "cons" cell at the front of the second list, $O(1)$ too). However, when the first list is depleted,

[2]In this particular context, $U_1 = \mathsf{List}\ T$ and $U_2 = \lfloor \mathsf{List}\ T \rfloor$, so "fillComp" has signature $\triangleleft\!\bullet : \lfloor \mathsf{List}\ T \rfloor \to \mathsf{DList}\ T \overset{1\uparrow}{\longrightarrow} \lfloor \mathsf{List}\ T \rfloor$

```
Queue T ≜ (List T)⊗(DList T)

singleton : T → Queue T
singleton x ≜ (Inr (x, Inl ()), alloc)

enqueue : Queue T → T → Queue T
q enqueue y ≜ q ▷ case (xs, ys) ↦ (xs, ys append y)

dequeue : Queue T → 1⊕(T⊗(Queue T))
dequeue q ≜ q ▷ case {
              (Inr (x, xs), ys) ↦ Inr (x, (xs, ys)),
              (Inl (), ys) ↦ (to_List ys) ▷ case {
                  Inl () ↦ Inl (),
                  Inr (x, xs) ↦ Inr (x, (xs, alloc))
              }
            }
```

Fig. 3. Queue implementation in equirecursive destination calculus

one has to transfer elements from the second list to the first one, and as such, has to reverse the second list, 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 maintain a quick **enqueue** operation (still $O(1)$), but remove the need for a **reverse** operation (as $\mathbf{to}_{\mathsf{List}}$ is $O(1)$ for difference lists). Nothing needs to change for the first list. The corresponding implementation is presented in Figure 3.

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

Replace by the empty queue

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

The **dequeue** operator is more complex though. It first checks if there is at least one element available in the front list. If there is, it extracts the element x by removing the first "cons" cell of the front list, 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 tail xs of the difference list turned into a list, and a fresh difference list given by **alloc**.

3 LIMITATIONS OF THE PREVIOUS APPROACH

Everything described above is in fact already possible in destination-passing style for Haskell as presented in [Bagrel 2024]. However, there is one fundamental limitation in [Bagrel 2024]: the inability to store destinations in destination-based data structures.

Indeed, that first approach of destination-passing style for Haskell can only be used to build non-linear data structures. More precisely, the "fillLeaf" operator ($\triangleleft$) can only take arguments with multiplicity ω. This is in fact a much stronger restriction than necessary ; the core idea is *just* to prevent any destination (which is always a linear resource) to appear somewhere in the right-hand side of "fillLeaf".

Don't start with the solution, start with the problem. Then we can explain how previous attempts

3.1 The problem with stored destinations

One core assumption of destination-passing style programming is that once a destination has been linearly consumed, 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 $\mathsf{T} = \lfloor \mathsf{T}' \rfloor$!

We have to mark the value being fed in a destination as linearly consumed, so that it cannot be both stored away (to be used later) and pattern-matched on/used in the current context. But that means we have to mark the destination $d : \lfloor \mathsf{T}' \rfloor$ as linearly consumed 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 consume a destination: feed it now with a value, or store it away and feed it later. The latter is a much weaker form of consumption, 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 assumption above doesn't hold in general case.

The issue is particularly visible when trying to give semantics to the **alloc′** operator with signature $\mathbf{alloc'} : (\lfloor \mathsf{T} \rfloor \to 1) \to \mathsf{T}$. It reads: "given a way of consuming a destination of type T, I'll return an object of type T". This is an operator we very much want in our system!

The morally correct semantics (in destination calculus pseudo-syntax) would be:

$$\mathbf{alloc'}\ (\lambda d \mapsto \mathrm{t}) \longrightarrow \mathbf{withTmpStore}\ \{h := \square\}\ \mathbf{do}\ \{\mathrm{t}[d := \to h]\ \fatsemi\ \mathbf{deref}\ \to h\}$$

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

$$\begin{aligned}
&\mathbf{alloc'}\ (\lambda d \mapsto d \triangleleft \mathsf{Inl} \triangleleft ()) \\
\longrightarrow\ &\mathbf{withTmpStore}\ \{h := \square\}\ \mathbf{do}\ \{\to h \triangleleft \mathsf{Inl} \triangleleft ()\ \fatsemi\ \mathbf{deref}\ \to h\} \\
\longrightarrow\ &\mathbf{withTmpStore}\ \{h := \mathsf{Inl}\ ()\}\ \mathbf{do}\ \{\mathbf{deref}\ \to h\} \\
\longrightarrow\ &\mathsf{Inl}\ ()
\end{aligned}$$

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

$$\begin{aligned}
&\mathbf{alloc'}\ (\lambda dd \mapsto \mathbf{alloc'}\ (\lambda d \mapsto dd \blacktriangleleft d)) \\
\longrightarrow\ &\mathbf{withTmpStore}\ \{hd := \square\}\ \mathbf{do}\ \{\mathbf{alloc'}\ (\lambda d \mapsto \to hd \blacktriangleleft d)\ \fatsemi\ \mathbf{deref}\ \to hd\} \\
\longrightarrow\ &\mathbf{withTmpStore}\ \{hd := \square\}\ \mathbf{do}\ \{\mathbf{withTmpStore}\ \{h := \square\}\ \mathbf{do}\ \{\to hd \blacktriangleleft \to h\ \fatsemi\ \mathbf{deref}\ \to h\}\ \fatsemi\ \mathbf{deref}\ \to hd\} \\
\longrightarrow\ &\mathbf{withTmpStore}\ \{hd := \to h\}\ \mathbf{do}\ \{\mathbf{withTmpStore}\ \{h := \square\}\ \mathbf{do}\ \{\mathbf{deref}\ \to h\}\ \fatsemi\ \mathbf{deref}\ \to hd\}
\end{aligned}$$

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 consumes d linearly. However, we see that because $\to h$ escaped to the parent scope by being stored in a destination of destination coming from the parent scope, the hole h has not been written to, and thus the inner expression $\mathbf{withTmpStore}\ \{h := \square\}\ \mathbf{do}\ \{\mathbf{deref}\ \to h\}$ cannot reduce in a meaningful way.

One could argue that the issue comes from the destination-filling primitive $\triangleleft$ returning unit instead of a special value of a distinct *effect* type. However, the same issue arise if we introduce a distinct type $\|$ for the effect of filling a destination ; there is always a way to cheat the system and make a destination escape to a parent scope. This distinct type for effects has in fact existed during the early prototypes of destination calculus, but we removed it as it doesn't solve the scope escape for destination and is indistinguishable in practice from the unit type.

3.2 Age control to prevent scope escape of destinations

The solution we chose is to instead track the age of destinations (as De-Brujin-like scope indices), and prevent a destination to escape into the parent scope when stored through age-control restriction on the typing rule of destination-filling primitives.

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.

I remember that we did wonder about that at some point, so it's worth preempting the question. Though I'd rather we found a slightly different example that doesn't exploit the return type of fill than a paragraph recounting our life story.
alloc & dd -> alloc & d -> dd <| d alloc & dd -> dd <| (,) & case

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 ∞ in practice.

Semiring addition $+$ is used to find the age of a variable or destination that is used in two different branches 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 on ages are presented in Figure 4.

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

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

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

Fig. 4. Tables for age operations

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

The main restriction to prevent parent scope escape is materialized the simplified typing rules of Figure 5.

Typing a destination $\rightarrow h$ alone requires $\rightarrow h$ to have age ν in the context. And when storing a value through a destination, the ages of the value's dependencies in the context must be one higher than the corresponding ages required to type the value alone (this is the meaning of $\uparrow \cdot \Theta_2$).

Such a rule system prevents in particular the previous faulty expression $\rightarrow hd \blacktriangleleft \rightarrow h$ where $\rightarrow hd$ originates from the context parent to the one of $\rightarrow h$.

4 BREADTH-FIRST TREE TRAVERSAL

The core example that showcases the power of destination-passing style programming with first-class destination 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 i.e., building the leaves first and going up to the root.

In [Bagrel 2024], 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, much 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 $(\langle \textit{input subtree} \rangle, \langle \textit{dest to output subtree} \rangle)$ of the queue is processed, and its children pairs are added back at the end of the queue to be processed later.

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

However, as evoked earlier in Section 3.1, The API presented in [Bagrel 2024] is not able to store linear data, and in particular destinations, in destination-based data structures. It is thus reliant on regular constructor-based Haskell data structures for destination storage.

$$\textsc{Ty-val-Dest}^\star \quad \frac{}{\to h :_\gamma \lfloor \mathsf{T} \rfloor \vdash \to h : \lfloor \mathsf{T} \rfloor}$$

$$\textsc{Ty-term-FillLeaf}^\star \quad \frac{\Theta_1 \vdash \mathsf{t} : \lfloor \mathsf{T} \rfloor \qquad \Theta_2 \vdash \mathsf{t}' : \mathsf{T}}{\Theta_1 + \uparrow \cdot \Theta_2 \vdash \mathsf{t} \triangleleft \mathsf{t}' : 1}$$

Fig. 5. Simplified typing rules for age control of destinations

$$\mathsf{Tree\ T} \triangleq 1 \oplus (\mathsf{T} \otimes ((\mathsf{Tree\ T}) \otimes (\mathsf{Tree\ T})))$$

$$\triangleleft\mathsf{Nil} : \lfloor_n \mathsf{Tree\ T} \rfloor \to 1$$
$$d \triangleleft \mathsf{Nil} \triangleq d \triangleleft \mathsf{Inl} \triangleleft ()$$

$$\triangleleft\mathsf{Node} : \lfloor_n \mathsf{Tree\ T} \rfloor \to \lfloor_n \mathsf{T} \rfloor \otimes (\lfloor_n \mathsf{Tree\ T} \rfloor \otimes \lfloor_n \mathsf{Tree\ T} \rfloor)$$
$$d \triangleleft \mathsf{Node} \triangleq (d \triangleleft \mathsf{Inr} \triangleleft (,)) \rhd \mathbf{case}\ (dv, dtlr) \mapsto (dv, dtlr \triangleleft (,))$$

$$\mathsf{Nat} \overset{\mathrm{rec}}{\triangleq} 1 \oplus \mathsf{Nat}$$

$$\mathbf{zero} : \mathsf{Nat}$$
$$\mathbf{zero} \triangleq \mathsf{Inl}\ ()$$

$$\mathbf{succ} : \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathbf{succ}\ x \triangleq \mathsf{Inr}\ x$$

Fig. 6. Boilerplate for breadth-first tree traversal

This is quite impractical as we would like to use the efficient, destination-based queue implementation from Section 2.2 to power up the breadth-first tree traversal implementation[3]. In our present work fortunately, thanks to the finer age-control mechanism, we can store linear resources in destination-based structures without any issue. Our system is in fact self-contained, as any structure, whatever the use for it be, can be built using a small core of destination-based primitives (and regular data constructors can be retrieved from destination-based primitives, see Section 5.4).

Figure 6 introduces a few extra tools needed for implementation of breadth-first tree traversal, while Figure 7 presents the actual implementation of the traversal. This implementation is as similar as possible to the one from [Bagrel 2024], as to make it easier to spot the few differences between the two systems.

The first important difference is that in the destination calculus implementation, the input tree of type $\mathsf{Tree\ T}_1$ is consumed linearly. The stateful transformer that is applied to each input node to get the output node value is also linear in its two arguments. The state has to be wrapped in an exponential $!_{1\infty}$ so that it can be extracted from the right side of the ampar at the end of the processing (with $\mathbf{from}_{\ltimes}$). We could imagine a more general version of the traversal, having no constraint on the state type, but necessitating a finalization function $\mathsf{S} \to !_{1\infty}\mathsf{S}'$ so that the final state can be returned.

There's way too many mode annotations here. If we are to do that we have to dedicate some space above to explaining the modes and what all the constructions around them mean.

The **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 feeds Nil to 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, processes the value of the node with the stateful transformer f, and continues the processing of the updated queue where children subtrees and their accompanying destinations have been enqueued.

mapAccumBFS is in charge of spawning the initial memory slot for the output tree together with the associated destination, and preparing the initial queue containing a single pair, made of the whole input tree and the aforementioned destination.

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

[3]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 : ((!1∞S) ⊸ T1 ⊸ (!1∞S)⊗T2) ων⟿ (!1∞S) ⊸ Queue (Tree T1⊗⌊Tree T2⌋) ⊸ (!1∞S)
go f st q ≜rec (dequeue q) ▷ case {
                  Inl () ↦ st,
                  Inr ((tree, dtree), q') ↦ tree ▷ case {
                      Inl () ↦ dtree ◃ Nil ⨟ go f st q',
                      Inr (x, (tl, tr)) ↦ (dtree ◃ Node) ▷ case
                          (dy, (dtl, dtr)) ↦ (f st x) ▷ case
                              (st', y)↦ dy ◀ y ⨟ go f st' (
                                  q' enqueue (tl, dtl) enqueue (tr, dtr)
                              )
                  }
              }

mapAccumBFS : ((!1∞S) ⊸ T1 ⊸ (!1∞S)⊗T2) ων⟿ (!1∞S) ⊸ Tree T1 ⊸ Tree T2⊗(!1∞S)
mapAccumBFS f st tree ≜ from⅋ (alloc ▷ map dtree ↦ go f st (singleton (tree, dtree)))

relabelDPS : Tree 1 ⊸ (Tree Nat)⊗(!1∞(!ωνNat))
relabelDPS tree ≜ mapAccumBFS
                      (λex ↦ λun ↦ un ⨟ ex ▷ case
                          E1∞ ex' ↦ ex' ▷ case1∞
                              Eων st ↦ (E1∞ (Eων (succ st)), st))
                      (E1∞ (Eων (succ zero)))
                       tree
```

Fig. 7. Breadth-first tree traversal in destination-passing style

by naturals $1 \dots |T|$ in breadth-first order. The higher-order function passed to **mapAccumBFS** is quite verbose: it must consume the previous node value (unit), using ⨟, then extract the state (representing the next natural number to attribute to a node) from its two nested exponential constructors, and finally return a pair, whose left side is the incremented natural wrapped back into its two exponential layers (new state), and whose right side is the plain natural representing the node value for the output subtree.

Two exponentials are needed here. The first one $!_{1\infty}$ is part of $\mathbf{from}_{⅋}$ contract, and ensures that the state cannot capture destinations, so that it can be returned at the end of the processing (as **go** runs under a **map** over an ampar). The second one $!_{\omega\nu}$ allows the natural number to be used in a non-linear fashion: it is used once in **succ** *st* to serve as the new state, and another time as the value for the output node. With a more general $\mathbf{from}_{⅋}$ operator, we would be able to use just one exponential layer $!_{\omega\infty}$ over the natural number to achieve the same result.

To Thomas: didn't you tell me you had plan of making something more general for this very case? I don't remember clearly. If so, don't forget to improve here

5 LANGUAGE SYNTAX

5.1 Introducing the *ampar*

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. In other terms, the bound *hole variable* cannot be pattern-matched on or used as a parameter of a function call. A hole abstraction is thus a weak form of linear lambda abstraction, which just moves a piece of data into a bigger data structure.

I'm [Arnaud] honestly uncomfortable at the quantity of mathematics written in this section without macros and completely uncorrelated from OTT. If we

In fact, the type of hole abstraction $(\mathsf{T}_1, \mathsf{T}_2)\mathsf{hfun}$ in Minamine's work shares a lot of similarity with the separating implication or *magic wand* $\mathsf{T}_1 \mathrel{-\!\!*} \mathsf{T}_2$ from separation logic: given a piece of memory matching description T_1, we obtain a (complete) piece of memory matching description T_2.

Now, in classical linear logic, we know we can transform linear implication $\mathsf{T}_1 \multimap \mathsf{T}_2$ into $\mathsf{T}_1{}^{\perp} \parr \mathsf{T}_2$. Doing so for the *wand* type $(\mathsf{T}_1, \mathsf{T}_2)\mathsf{hfun}$ or $\mathsf{T}_1 \mathrel{-\!\!*} \mathsf{T}_2$ gives $\lfloor \mathsf{T}_1 \rfloor \mathbin{\widehat{\parr}} \mathsf{T}_2$, where $\lfloor \bullet \rfloor$ is memory negation, and $\mathbin{\widehat{\parr}}$ is a memory *par* (weaker than the CLL *par* that allows more "interaction" of its two sides).

Transforming the hole abstraction from its original implication form to a *par* form let us consider the type $\lfloor \mathsf{T}_1 \rfloor$ of *sink* or *destination* of T_1 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 $\mathbin{\widehat{\parr}}$ to ampar $\ltimes$. In CLL, the cut rule states that given $\mathsf{T}_1 \parr \mathsf{T}_2$, we can free up T_1 by providing an eliminator of T_2, or free up T_2 by providing an eliminator of T_1. The eliminator of T can be $\mathsf{T}^{\perp}$, or $\mathsf{T}^{\perp^{-1}} = \mathsf{T}'$ if T is already of the form $\mathsf{T}'^{\perp}$. In a classical setting, thanks to the involutive nature of negation $\bullet^{\perp}$, the two potential forms of the eliminator of T are equal.

In destination calculus though, we don't have an involutive memory negation $\lfloor \bullet \rfloor$. If we are provided with a destination of destination $\rightarrow h' : \lfloor \lfloor \mathsf{T} \rfloor \rfloor$, we know that some structure is expecting to store a destination of type $\lfloor \mathsf{T} \rfloor$. If ever that structure is consumed, then the destination stored inside will have to be fed with a value (remember we are in a linear calculus). So if we allocate a new memory slot of type $\boxed{h} : \mathsf{T}$ and its linked destination $\rightarrow h : \lfloor \mathsf{T} \rfloor$, and write $\rightarrow h$ to the memory slot pointed to by $\rightarrow h'$, then we can get back a value of type T at $\boxed{h}$ if ever the structure pointed to by $\rightarrow h'$ is consumed. Thus, a destination of destination is only equivalent to the promise of an eventual value, not an immediate usable one.

As a result, in destination calculus, we cannot have the same kind of cut rule as in CLL. This is, in fact, the part of destination calculus that was the hardest to design, and the source of a lot of early errors. For a destination of type $\lfloor \mathsf{T} \rfloor$, both storing it through a destination of destination $\lfloor \lfloor \mathsf{T} \rfloor \rfloor$ or using it to store a value of type T constitute a linear use of the destination. But only the latter is a genuine consumption in the sense that it guarantees that the hole associated to the destination has been written to! Storing away the destination of type $\lfloor \mathsf{T} \rfloor$ originating from $\mathsf{T} \mathbin{\widehat{\parr}} \lfloor \mathsf{T} \rfloor$ (through a destination of destination of type $\lfloor \lfloor \mathsf{T} \rfloor \rfloor$) should not allow to free up the T, as it would in a CLL-like setting.

However, we can recover a memory abstraction that is usable in practice if we know the nature of an memory par side:

- if the memory par side is a value made only of inert elements and destinations (negative polarity), then we can pattern-match/**map** on it, but we cannot store it away to free up the other side;
- if the memory par side is a value made only of inert elements and holes (positive polarity), then we can store it away in a bigger struct and free up the associated destinations (this is not an issue as the bigger struct will be locked by an memory par too), but we cannot pattern-match/**map** on it as it (may) contains holes;
- if one memory par side is only made of inert elements, we can in fact convert the memory par to a pair, as the memory par doesn't have any form of interaction between its sides.

It is important to note that the type of an memory par side is not really enough to determine the nature of the side, as a hole of type T and and inert value of type T are indistinguishable at the type level.

So we introduced a more restricted form of memory par, named *ampar* ($\ltimes$), for *asymmetrical memory par*, in which:

- the left side is made of inert elements (normal values or destinations from previous scopes) and/or holes if and only if those holes are compensated by destinations on the right side;
- the right side is made of inert elements and/or destinations.

As the right side cannot contain any holes, it is always safe to pattern-match or **map** on it. Because the left side cannot contain destinations from the current scope, it is always safe to store it away in a bigger struct and release the right side.

Finally, it is enough to check for the absence of destinations in the right side (which we can do easily just by looking at its type) to convert an *ampar* to a pair, as any remaining hole on the left side would be compensated by a destination on the right side.

Destinations from previous scopes are inert. In destination calculus, scopes are delimited by the **map** operation over ampars. Anytime a **map** happens, we enter a new scope, and any preexisting destination or variable see its age increased by one ($\uparrow$). As soon as a destination or variable is no longer of age 0 (ν), it cannot be used actively but only passively (e.g. it cannot be applied if it is a function, or used to store a value if it is a destination, but it can be stored away in a dest, or pattern-matched on).

This is a core feature of the language that ensures part of its safety.

5.2 Names and variables

The destination calculus uses two classes of names: regular variable names x, y, and *hnames* h, h_1, h_2 which are identifiers for a memory cell that hasn't been written to yet, as illustrated in Figure 8.

Hole names are represented by natural numbers under the hood, so they 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 hname encountered so far in the program ; this corresponds to finding the next unused memory cell in which to write new data.

We sometimes need to keep track of hnames bound by a particular runtime value or evaluation context, hence we also define sets of hnames H, H_1, H_2 ....

Shifting all hnames in a set by a given offset h' is denoted $H \pm h'$. We also define a conditional shift operation $[H \pm h']$ which shifts each hname appearing in the operand to the left of the brackets by h' if this hname is also member of H. This conditional shift can be used on a single hname, a value, or a typing context.

5.3 Term and value core syntax

Destination calculus is based on linear simply-typed λ-calculus, with built-in support for sums, pairs, and exponentials. The syntax of terms, which is presented in Figure 9 is quite unusual, as we need to introduce all the tooling required to manipulate destinations, which constitute the primitive way of building a data structures for the user.

In fact, the grammatical class of values v, presented as a subset of terms t, could almost be removed completely from the user syntax, and just used as a denotation for runtime data structures. We only need to keep the *ampar* value ${}_{\{h\}}\langle \boxed{h}_{\wedge} \to h\rangle$ as part of the user syntax as a way to spawn a fresh memory cell to be later filled using destination-filling primitives (see **alloc** in Section 5.4).

Pattern-matching on every type of structure (except unit) is parametrized by a mode m to which the scrutinee is consumed. The variables which bind the subcomponents of the scrutinee then inherit this mode. In particular, this choice crystalize the equivalence $!_{\omega a}(\mathsf{T}_1 \otimes \mathsf{T}_2) \simeq (!_{\omega a}\mathsf{T}_1) \otimes (!_{\omega a}\mathsf{T}_2)$, which is not part of intuitionistic linear logic, but valid in Linear Haskell[Bernardy et al. 2018]. We

var, *x*, *y*, *d*, *dd*, *un*, *xs*, *ys*, *ex*, *st*, *tree*, *tl*, *tr*, *dtree*, *f*, *dh*, *dt*, *dx*, *dy*, *dxs*, *dys*, *dv*, *dtlr*, *dtl*, *dtr*, *q*, *tok*				Variable names
$hname$, h, hd	::=			Hole (or destination) name, represented by a natural number
	|	$h \dotplus h'$	M	
	|	$h[H \dotminus h']$	M	Shift by h' if $h \in H$
	|	$\mathbf{max}(H)$	M	Maximum of a set of hole names
$hnames$, H	::=			Set of hole names
	|	$\{h_1, .., h_k\}$		
	|	$H_1 \cup H_2$	M	Union of sets
	|	$H \dotminus h'$	M	Shift all names from H by h'.
	|	$\mathbf{hnames}(\Gamma)$	M	Hole names bound by the typing context Γ
	|	$\mathbf{hnames}(\mathrm{C})$	M	Hole names bound by the evaluation context C

Fig. 8. Grammar for variable, hole and destination names

omit the mode annotation on **case** statements and lambda abstractions when the mode in question is the multiplicative neutral element 1_ν of the mode semiring.

map is the main primitive to operate on an *ampar*, which represents an incomplete data structure whose building is in progress. **map** binds the right-hand side of the *ampar* — the one containing destinations of that *ampar* — to a variable, allowing those destinations to be operated on by destination-filling primitives. The left-hand side of the *ampar* is inaccessible as it is being mutated behind the scenes by the destination-filling primitives.

$\mathbf{to}_{\ltimes}$ embeds an already completed structure in an *ampar* whose left side is the structure, and right side is unit. We have an operator "fillComp" ($\triangleleft\bullet$) allowing to compose two *ampar*s by writing the root of the second one to a destination of the first one, so by throwing $\mathbf{to}_{\ltimes}$ to the mix, we can compose an *ampar* with a normal (completed) structure (see the sugar operator "fillLeaf" ($\triangleleft$) in Section 5.4).

$\mathbf{from}_{\ltimes}$ is used to convert an *ampar* to a pair, when the right side of the *ampar* is an exponential of the form $\mathrm{E}_{1\infty}\ \mathrm{v}$. Indeed, when the right side has such form, it cannot contains destinations (as destinations always have a finite age), thus it cannot contain holes in its left side either (as holes on the left side are always compensated 1:1 by a destination on the right side). As a result, it is valid to convert an *ampar* to a pair in these circumstances. $\mathbf{from}_{\ltimes}$ is in particular used to extract a structure from its *ampar* building shell when it is complete (see the sugar operator $\mathbf{from'}_{\ltimes}$ in Section 5.4).

The remaining term operators $\triangleleft()$, $\triangleleft\mathsf{Inl}$, $\triangleleft\mathsf{Inr}$, $\triangleleft\,\mathrm{E}_m$, $\triangleleft(,)$, $\triangleleft(\lambda x\,{}_m \mapsto \mathrm{u})$ are all destination-filling primitives. They write a layer of value/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).

Values. There are two important things to note on the value class.

First, a variable cannot contains any free variable. This will be better visible in the typing rule for function value form Ty-val-Fun, but a value only admits hole and destination type bindings in its typing context, no variable binding. This is quite useful for substitution lemmas, as no undesired capture can happen.

Secondly, values are allowed to have holes inside (represented by $\boxed{h}$, $\boxed{h_1}$, $\boxed{h_2}$...), but a value used as a term isn't allowed to have any free hole (i.e. a hole that is not compensated by an associated destination inside an ampar). This is enforced by the typing context Δ meaning "destination-only" in the rule Ty-term-Val.

term, t, u	::=		Term
	|	v	Value
	|	x	Variable
	|	$t'\ t$	Application
	|	$t \fatsemi u$	Pattern-match on unit
	|	$t \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\}$	Pattern-match on sum
	|	$t \triangleright \mathbf{case}_m\ (x_1\ ,\ x_2) \mapsto u$	Pattern-match on product
	|	$t \triangleright \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u$	Pattern-match on exponential
	|	$t \triangleright \mathbf{map}\ x \mapsto t'$	Map over the right side of ampar
	|	$\mathbf{to}_{\ltimes}\ u$	Wrap into a trivial ampar
	|	$\mathbf{from}_{\ltimes}\ t$	Convert ampar to a pair
	|	$t \triangleleft ()$	Fill destination with unit
	|	$t \triangleleft \mathsf{Inl}$	Fill destination with left variant
	|	$t \triangleleft \mathsf{Inr}$	Fill destination with right variant
	|	$t \triangleleft \mathsf{E}_m$	Fill destination with exponential constructor
	|	$t \triangleleft (,)$	Fill destination with product constructor
	|	$t \triangleleft (\lambda x\ {}_m\!\mapsto u)$	Fill destination with function
	|	$t \triangleleft\!\bullet\ t'$	Fill destination with root of other ampar
	|	$t[x := v]$ M	

val, v	::=		Value
	|	$\boxed{h}$	Hole
	|	$\rightarrow h$	Destination
	|	()	Unit
	|	${}^{\mathbf{v}}\lambda x\ {}_m\!\mapsto u$	Function with no free variable
	|	$\mathsf{Inl}\ v$	Left variant for sum
	|	$\mathsf{Inr}\ v$	Right variant for sum
	|	$\mathsf{E}_m\ v$	Exponential
	|	$(v_1\ ,\ v_2)$	Product
	|	$H\langle v_2\ ,_\wedge v_1\rangle$	Ampar
	|	$v[H \doteq h']$ M	Shift hole names inside v by h' if they belong to H.

Fig. 9. Grammar for terms and values

5.4 Syntactic sugar for constructors and commonly used operations

As we said in section 5.3, the grammatical class of values is mostly used for runtime only; in particular, data constructors in the value class can only take other values as arguments, not terms (this help us ensure that no free variable can appear in a value). Thus we introduce syntactic for data constructors taking arbitrary terms as parameters (as we often find in functional programming languages) using destination-filling primitives in Figure 10.

$\mathbf{from}_{\ltimes}{}'$ is a simpler variant of $\mathbf{from}_{\ltimes}$ that allows to extract the right side of an ampar when the right side has been fully consumed. We implement it in terms of $\mathbf{from}_{\ltimes}$ 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.

All the desugarings are presented in Figure 11.

sterm, $\tilde{t}$	::=			Syntactic sugar for terms
	|	**alloc**	M	Evaluate to a fresh new ampar
	|	$t \blacktriangleleft t'$	M	Fill destination with supplied term
	|	$\mathbf{from}'_{\ltimes}\ t$	M	Extract left side of ampar when right side is unit
	|	$\lambda x\ {}_{m}\!\mapsto u$	M	Allocate function
	|	$\mathsf{Inl}\ t$	M	Allocate left variant
	|	$\mathsf{Inr}\ t$	M	Allocate right variant
	|	$\mathsf{E}_m\ t$	M	Allocate exponential
|	(t_1, t_2)	M		Allocate product

Fig. 10. Syntactic sugar forms for terms

$$\mathbf{alloc} \triangleq \ {}_{\{1\}}\langle \boxed{1}\ {}_{\wedge} \longrightarrow 1\rangle$$

$$\mathbf{from}'_{\ltimes}\ t \triangleq (\mathbf{from}_{\ltimes}\ (t \triangleright \mathbf{map}\ un \mapsto un\ \fatsemi\ \mathsf{E}_{1\infty}\ ())) \triangleright \mathbf{case}\ (st, ex) \mapsto ex \triangleright \mathbf{case}\ \mathsf{E}_{1\infty}\ un \mapsto un\ \fatsemi\ st$$

$$\lambda x\ {}_{m}\!\mapsto u \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft (\lambda x\ {}_{m}\!\mapsto u)\ )$$

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

$$t \blacktriangleleft t' \triangleq t \triangleleft\!\bullet (\mathbf{to}_{\ltimes}\ t')$$

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

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

$$\mathsf{E}_m\ t \triangleq \mathbf{from}'_{\ltimes}\ (\ \mathbf{alloc} \triangleright \mathbf{map}\ d \mapsto d \triangleleft \mathsf{E}_m \blacktriangleleft t\ )$$

Fig. 11. Desugaring of syntactic sugar forms for terms

6 TYPE SYSTEM

6.1 Types, modes, and typing contexts

λ_d is equipped with a first-order type system, featuring modal function types and modal boxing, in addition to unit (1), product ($\otimes$) and sum ($\oplus$) types. Modes have two axes — multiplicity (i.e. linear/non-linear), and age control — and they take place on variable bindings in typing contexts, and on function arrows, but are not part of the type itself. The type system is highly inspired from[Abel and Bernardy 2020] and Linear Haskell [Bernardy et al. 2018]. In particular, it uses the same additive/multiplicative approach on contexts for linearity and age enforcement.

The grammar of types, modes, and typing context for λ_d is presented in Figure 12. As said earlier, the function arrow ${}_{m}\!\longrightarrow$ and exponential connective $!_m$ are both parametrized by a mode m; we omit that mode annotation when the mode in question is the multiplicative neutral element 1_ν of the 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].

6.2 Typing of terms and values

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* $\boxed{h} :_n \mathsf{T}$ and

type, T, U, S	::=		Type
	|	1	Unit
	|	$T_1 \oplus T_2$	Sum
	|	$T_1 \otimes T_2$	Product
	|	$!_m T$	Exponential
	|	$U \ltimes T$	Ampar
	|	$T \multimap_m U$	Function
	|	$\lfloor_m T \rfloor$	Destination
mode, m, n	::=		Mode (Semiring)
	|	pa	Pair of a multiplicity and age
	|	💀	Error case (incompatible types, multiplicities, or ages)
mul, p	::=		Multiplicity (Semiring, first component of mode)
	|	1	Linear use
	|	ω	Non-linear use
age, a	::=		Age (Semiring, second component of mode)
	|	ν	Born now
	|	$\uparrow$	One scope older
	|	∞	Infinitely old / static
ctx, Ω, Γ, Δ, Θ	::=		Typing context
	|	$x :_m T$	Variable typing binding
	|	$\boxed{h} :_n T$	Hole typing binding
	|	$\rightarrow h :_m \lfloor_n T \rfloor$	Destination typing binding
	|	$m \cdot \Omega$ M	Multiply the leftmost mode of each binding by m
|	$\Omega_1 + \Omega_2$ M		Sum (incompatible bindings get tagged with 💀)
|	Ω_1, Ω_2 M		Disjoint sum
|	$\rightarrow^{-1}\Delta$ M		Transforms dest bindings into a hole bindings
|	$\Omega[H \pm h']$ M		Shift hole/dest names by h' if they belong to H

Fig. 12. Types, modes, and typing contexts

destination bindings $\rightarrow h :_m \lfloor_n T \rfloor$ in addition to the variable bindings $x :_m T$ that usually populates typing contexts.

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 filled in this destination.

Figure 13 presents the typing rules for terms, and rules for syntactic sugar forms that have been derived from term rules and proven formally too. Figure 14 presents the typing rules for values of the language. In every figure,

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

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

$$\textsc{Ty-term-Val}\ \frac{\texttt{DisposableOnly}\ \Theta \qquad \Delta\ \mathbin{\overset{\nu}{\vdash}}\ v : T}{\Theta, \Delta \vdash v : T} \qquad \textsc{Ty-term-Var}\ \frac{\texttt{DisposableOnly}\ \Theta \qquad 1\nu <: m}{\Theta,\ x :_m T \vdash x : T} \qquad \textsc{Ty-term-App}\ \frac{\Theta_1 \vdash t : T \qquad \Theta_2 \vdash t' : T \xrightarrow{m} U}{m{\cdot}\Theta_1 + \Theta_2 \vdash t'\ t : U}$$

$$\textsc{Ty-term-PatU}\ \frac{\Theta_1 \vdash t : 1 \qquad \Theta_2 \vdash u : U}{\Theta_1 + \Theta_2 \vdash t \fatsemi u : U} \qquad \textsc{Ty-term-PatS}\ \frac{\Theta_1 \vdash t : T_1 \oplus T_2 \qquad \Theta_2,\ x_1 :_m T_1 \vdash u_1 : U \qquad \Theta_2,\ x_2 :_m T_2 \vdash u_2 : U}{m{\cdot}\Theta_1 + \Theta_2 \vdash t \rhd \mathbf{case}_m\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\} : U}$$

$$\textsc{Ty-term-PatP}\ \frac{\Theta_1 \vdash t : T_1 \otimes T_2 \qquad \Theta_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{m{\cdot}\Theta_1 + \Theta_2 \vdash t \rhd \mathbf{case}_m\ (x_1\ ,\ x_2) \mapsto u : U} \qquad \textsc{Ty-term-PatE}\ \frac{\Theta_1 \vdash t : !_n T \qquad \Theta_2,\ x :_{m\cdot n} T \vdash u : U}{m{\cdot}\Theta_1 + \Theta_2 \vdash t \rhd \mathbf{case}_m\ \mathsf{E}_n\ x \mapsto u : U}$$

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

$$\textsc{Ty-term-FillU}\ \frac{\Theta \vdash t : \lfloor_n 1 \rfloor}{\Theta \vdash t \triangleleft () : 1} \qquad \textsc{Ty-term-FillL}\ \frac{\Theta \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Theta \vdash t \triangleleft \mathsf{Inl} : \lfloor_n T_1 \rfloor} \qquad \textsc{Ty-term-FillR}\ \frac{\Theta \vdash t : \lfloor_n T_1 \oplus T_2 \rfloor}{\Theta \vdash t \triangleleft \mathsf{Inr} : \lfloor_n T_2 \rfloor} \qquad \textsc{Ty-term-FillP}\ \frac{\Theta \vdash t : \lfloor_n T_1 \otimes T_2 \rfloor}{\Theta \vdash t \triangleleft (,) : \lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor}$$

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

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

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

$$\textsc{Ty-sterm-Fun}\ \frac{\Theta_2,\ x :_m T \vdash u : U}{\Theta_2 \vdash \lambda x_m \mapsto u : T \xrightarrow{m} U} \qquad \textsc{Ty-sterm-Left}\ \frac{\Theta_2 \vdash t : T_1}{\Theta_2 \vdash \mathsf{Inl}\ t : T_1 \oplus T_2} \qquad \textsc{Ty-sterm-Right}\ \frac{\Theta_2 \vdash t : T_2}{\Theta_2 \vdash \mathsf{Inr}\ t : T_1 \oplus T_2} \qquad \textsc{Ty-sterm-Exp}\ \frac{\Theta_2 \vdash t : T}{m{\cdot}\Theta_2 \vdash \mathsf{E}_m\ t : !_m T}$$

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

Fig. 13. Typing rules for terms and syntactic sugar

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 13 never explicitly mention a destination binding; only variable bindings. Only at runtime such variables will be substituted by destinations having a matching type and mode (that is why terms cannot

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

$$\textsc{Ty-val-Hole}\ \frac{}{\boxed{h} :_{1\nu} T \vdash \boxed{h} : T} \qquad \textsc{Ty-val-Dest}\ \frac{}{\to h :_{1\nu} \lfloor_n T \rfloor \vdash \to h : \lfloor_n T \rfloor} \qquad \textsc{Ty-val-Unit}\ \frac{}{\cdot \vdash () : 1}$$

$$\textsc{Ty-val-Fun}\ \frac{\Delta,\ x :_m T \vdash u : U}{\Delta \vdash \lambda x \,_m\!\mapsto u : T \,_m\!\to U}$$

$$\textsc{Ty-val-Left}\ \frac{\Gamma \vdash v_1 : T_1}{\Gamma \vdash \mathsf{Inl}\, v_1 : T_1 \oplus T_2} \qquad \textsc{Ty-val-Right}\ \frac{\Gamma \vdash v_2 : T_2}{\Gamma \vdash \mathsf{Inr}\, v_2 : T_1 \oplus T_2}$$

$$\textsc{Ty-val-Prod}\ \frac{\Gamma_1 \vdash v_1 : T_1 \qquad \Gamma_2 \vdash v_2 : T_2}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2) : T_1 \otimes T_2} \qquad \textsc{Ty-val-Exp}\ \frac{\Gamma \vdash v' : T}{n\cdot\Gamma \vdash \mathsf{E}_n\, v' : !_n T}$$

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

Fig. 14. Typing rules for values

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.

Let's focus on a few particularities of that system.

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 $\overset{\blacktriangledown}{\vdash}$).

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 such:

$$\begin{cases} pa <: p'a' \iff p \overset{\mathbf{p}}{<:} p' \wedge a \overset{\mathbf{a}}{<:} a' \\ m <: \text{☠} \end{cases}$$

$$\begin{cases} 1 \overset{\mathbf{p}}{<:} 1 \\ p \overset{\mathbf{p}}{<:} \omega \end{cases}$$

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

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

Rules Ty-term-App, Ty-term-PatS, Ty-term-PatP and Ty-term-PatE are all parametrized by a mode m by which the typing context Θ_1 of the argument is multiplied. These rules otherwise follows closely from [Bernardy et al. 2018].

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'. The core idea is that **map** creates a new scope for x and t', so anything coming from the current 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 $t \triangleright \mathbf{map}\ x \mapsto t'$ 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 15.

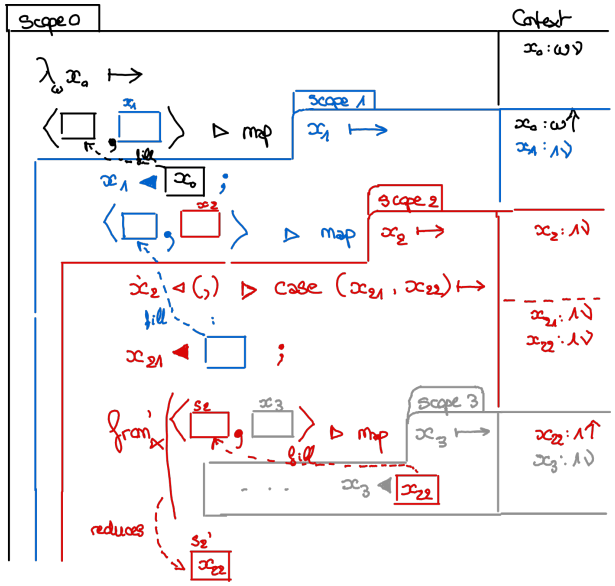

Fig. 15. Scope rules for **map** in λ_d

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 mapped over. When filling a destination (e.g. $x_1 \blacktriangleleft x_0$ in the figure), the right operand must be from a scope one $\uparrow$ 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 from Figure 13, or its simpler variant, Ty-sterm-FillLeaf confirm this intuition. The left operand of these operators must be a destination that types in the ambient 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 new scope for its body, "fillComp" ($\blacktriangleleft$)/"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 the lambda abstraction in Ty-term-FillF.

When using $\mathbf{from}'_{\ltimes}$, the left of an ampar is extracted to the ambient scope (as seen at the bottom of the figure with x_{22}). This is the fundamental reason why the left of an ampar has to share that same scope! When an ampar is complete and disposed of with the more general $\mathbf{from}_{\ltimes}$ however, we extract both sides of the ampar to the ambient scope, even though the right side is not part of that scope! This is only safe to do because the right side is required to have type $!_{1\infty} T$, which means it is scope-insensitive (it cannot contain any scope-controlled resource).

Typing of values $\vdash$. The typing of runtime values is where hole and destination bindings appears.

Hole bindings and destination bindings of the same hname are meant to compensate each other in the typing context, a bit like how matter (positive polarity, hole) and antimatter (negative polarity, destination) annihilate each other. That way, the typing context of a term can stay constant during reduction, even when destination-filling primitives are evaluated to build up data structures, as those linearly consume a destination and write to a hole at the same time which makes both disappear.

However, that annihilation between a destination and a hole binding having the same hname in the typing tree is only allowed to happen around an ampar, as it is the ampar connective that bind the two polarities of a name together (the names bound are actually stored in a set H on the ampar value $_H\langle v_2 \wedge v_1\rangle$). In fact, an ampar can be seen as a sort of lambda-abstraction, whose body (containing holes instead of variables) and sink/input site are split on two sides, and magically interconnected through the ampar connective.

Here, just speak how Ty-val-Prod and Ty-val-Ampar are different. In Coq, a context where both hole/dest for same name is not representable

Thus, the sum of typing contexts, used in almost all rules, results in an erroneous context when the input operands contains the same hname but in different polarities: $\{\boxed{h} :_n T\} + \{\to h :_{1\nu} \lfloor_n T\rfloor\} = \{\boxed{h} :_\omega 1\}$ (the fact that the result is a hole binding instead of a destination binding is purely arbitrary ; the only important bit here is to have an "invalid" mode on the result binding). An individual context is also not allowed to contain a same hname in two different bindings. Instead, in the only position where the annihilation is allowed to happen, that is to say the Ty-val-Ampar rule, we explicitly identify and remove the parts of input contexts that can interact and annihilate each other, in the form of Δ in the right side context and $\to^{-1}\Delta$ for the left one.

$\to^{-1}$ is a point-wise operation on typing bindings of a context where:

$$\begin{cases} \to^{-1}(\to h :_{1\nu} \lfloor_n T\rfloor) &= \boxed{h} :_n T \\ \to^{-1}(n :_m T) &= n :_\omega 1 \quad \text{otherwise} \end{cases}$$

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

It might be time to discuss what modes mean for hole and destination bindings.

A hole binding $\boxed{h} :_n T$ has only one mode, n, that 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). 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 9 that other forms of intrinsic linearity can be added to the langage for practical reasons. 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.

change to inner/outer one

On the other hand, a destination binding $\to h :_m \lfloor_n T\rfloor$ has two modes. The left-most one, m, tells how the destination can be used as a passive value, e.g. if the destination get stored in a destination of destination or passed as a function argument. The right-most one, n, corresponds to the mode of its associated hole, and thus how it behaves as an active entity, when filled with a value. We'll see in an instant how a binding can *grow old*, but let's just remember for now that a destination can only be filled with a value when $\mathbf{age}(m) = \nu$. As soon at it gets older (only the mode m is affected, the mode n of its associated hole never changes), it behaves like a passive normal value and can only be stored, passed around, or returned.

A closed term can never have a destination binding with left-most multiplicity ω or age ∞ somewhere in the typing tree. They can only be linear and have a finite age. Indeed, only Ty-val-Ampar (or more precisely, Ty-ectxs-OpenAmpar-Foc) adds new destination bindings to a typing context, and those initially have mode 1ν, and can only grow old later by increments of $\uparrow$ (but can never reach age ∞). We have formally proved that this property holds during execution. So any typing context of the form $\to h :_{\omega a} \lfloor_n \mathsf{T} \rfloor$, Ω or $\to h :_{p\infty} \lfloor_n \mathsf{T} \rfloor$, Ω is never satisfiable.

Values type in a typing context Γ made only of hole and destination bindings. The absence of free variables in values make it easier to prove substitution properties (as we will see in Section 7.3, we perform substitutions not only in terms, but also in evaluation contexts sometimes).

Rules Ty-val-Hole and Ty-val-Dest indicates that a hole or destination must have left-most mode 1ν in the typing context to be used (except when a destination is stored away, as we will see later). Rules for unit, left and right variants, and product are straightforward.

revisit this if we allow weakening for dests

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 $\mathtt{E}_2\ \mathrm{v}$, that corresponds to two uses of v (in a system with such a mode), will use v' four times. The mode product on context $n \cdot \Gamma$ in the exponential rule has a somewhat different effect on hole and destination bindings:

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

On hole bindings $\boxed{h} :_{n'} \mathsf{T}$, it affects the mode n' of the value the hole can receive, as we said earlier. On destination bindings $\to h :_m \lfloor_{n'} \mathsf{T} \rfloor$, on the other hand, it affects the mode m of the destination viewed as a value (the left-most one), but not the mode of the value it can be filled with.

Do we need more details on what are the consequences of that difference?

Rule Ty-val-Fun indicates that (value level) lambda abstractions can only contain (captured) destinations, they 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 Δ, $x :_m \mathsf{T}$ 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 $\lambda x \mapsto \mathbf{from}'_{\ltimes}\ (\mathbf{alloc} \rhd \mathbf{map}\ d \mapsto d \triangleleft (\lambda y \mapsto x\ \mathbf{concat}\ y))$, 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.

The last important rule, Ty-val-Ampar, is probably the most complex one. The left side v_2 is the data structure under construction, that may contain holes (in $\to^{-1}\Delta_3$), but also stored destinations from other scopes (in Δ_2). The right side v_1 is an arbitrary value, that we can manipulate using **map**, and that must contain all destinations matching the holes of v_2 (these destinations are represented by Δ_3). The right side v_1 may also contain stored destinations from other scopes, in $1\uparrow \cdot \Delta_1$ (in which case those cannot be filled in that scope, because they are at least of age $\uparrow$, and thus can only be stored or moved around). The resulting typing context for the ampar is Δ_1, Δ_2; it doesn't mention $\to^{-1}\Delta_3$ and Δ_3 anymore as they annihilate each other, as we explained previously. Δ_1 and Δ_2, by convention, are made only of destination bindings, and thus we know the annihilation (and thus binding) between holes and destinations for this ampar is indeed maximal.

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 $\to^{-1}\Delta_3$ is a valid typing context, so are not really a new restriction on Δ_3. They are mostly used to ease the mechanical proof of type safety for the system.

ectx, c	::=			Evaluation context component
	|	$t' \ \square$		
	|	$\square \ v$		
	|	$\square \fatsemi u$		
	|	$\square \triangleright \mathbf{case}_m \ \{\mathsf{Inl}\ x_1 \mapsto u_1\,,\ \mathsf{Inr}\ x_2 \mapsto u_2\}$		
	|	$\square \triangleright \mathbf{case}_m \ (x_1\,,\ x_2) \mapsto u$		
	|	$\square \triangleright \mathbf{case}_m \ \mathsf{E}_n \ x \mapsto u$		
	|	$\square \triangleright \mathbf{map}\ x \mapsto t'$		
	|	$\mathbf{to}_{\ltimes} \ \square$		
	|	$\mathbf{from}_{\ltimes} \ \square$		
	|	$\square \triangleleft ()$		
	|	$\square \triangleleft \mathsf{Inl}$		
	|	$\square \triangleleft \mathsf{Inr}$		
	|	$\square \triangleleft \mathsf{E}_m$		
	|	$\square \triangleleft (,)$		
	|	$\square \triangleleft (\lambda x \ {}_m\!\mapsto u)$		
	|	$\square \triangleleft\!\bullet \ t'$		
	|	$v \triangleleft\!\bullet \ \square$		
	|	$\overset{\mathrm{op}}{H}\langle v_2 \ {}_\wedge\!\square\rangle$		Open ampar, binding hole names in the next comp
ectxs, C	::=			Evaluation context stack
	|	$\square$		Represent the empty stack / "identity" evaluation c
	|	$C \circ c$		Push c on top of C
	|	$C[h :=_H \ v]$	M	Fill h in C with value v (that may contain holes)

Fig. 16. Grammar for evaluation contexts

7 EVALUATION CONTEXTS AND SEMANTICS

The semantics of λ_d are given using small-step reductions on a pair $C[t]$ of an evaluation context C (represented by a stack) determining a focusing path, and a term t under focus. Such a pair $C[t]$ is called a *command*, and represents a running program.

7.1 Evaluation contexts forms

The grammar of evaluation contexts is given in Figure 16. An evaluation context C is the composition of an arbitrary number of focusing components $c_1, c_2 \ldots$. We chose to represent this composition explicitly using a stack, instead of a meta-operation that would only let us access its final result. As a result, focusing and defocusing operations are made explicit in the semantics, resulting in a more verbose but simpler proof. It is also easier to imagine how to build a stack-based interpreter for such a language.

Focusing components are all directly derived from the term syntax, except for the "open ampar" focus $\overset{\mathrm{op}}{H}\langle v_2 \ {}_\wedge\!\square\rangle$. 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 $C[h :=_H \ v]$ 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.

7.2 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 $\Theta \vdash t : T$ 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 $\Delta \dashv C : T \rightarrowtail U_0$ is subtractive : adding the focus component $t' \ \square$ 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 $C[t]$ are presented in Figure 17.

An evaluation context has a pseudo-type $T \rightarrowtail U_0$, 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 $T \rightarrowtail U_0$ 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 $m \cdot \Delta_1 + \Delta_2$ in the premise for C corresponds to $m \cdot \Theta_1 + \Theta_2$ in the conclusion of Ty-term-PatP in Figure 13;
- the typing context $\Delta_2,\ x_1 :_m T_1,\ x_2 :_m T_2$ in the premise for term u corresponds to the typing context $\Theta_2,\ x_1 :_m T_1,\ x_2 :_m T_2$ for the same term in Ty-term-PatP;
- the typing context Δ_1 in the conclusion for $C \circ (\square \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u)$ 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 $C[t]$ (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 $\cdot \vdash t : T$ as a well-typed command using the identity evaluation context: $t \simeq \square[t]$ and we thus have $\vdash \square[t] : T$ where $\Delta = \cdot$ (the empty context).

7.3 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);

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

$$\textsc{Ty-ectxs-Id}\ \frac{}{\cdot \dashv \square : U_0 \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-App-Foc1}\ \frac{m\cdot\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2 \vdash t' : T \multimap_m U}{\Delta_1 \dashv C \circ (t'\ \square) : T \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-App-Foc2}\ \frac{m\cdot\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_1 \vdash v : T}{\Delta_2 \dashv C \circ (\square\ v) : (T \multimap_m U) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-PatU-Foc}\ \frac{\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2 \vdash u : U}{\Delta_1 \dashv C \circ (\square \fatsemi u) : 1 \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-PatS-Foc}\ \frac{m\cdot\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x_1 :_m T_1 \vdash u_1 : U \qquad \Delta_2,\ x_2 :_m T_2 \vdash u_2 : U}{\Delta_1 \dashv C \circ (\square \triangleright \mathbf{case}_m \{\mathsf{Inl}\, x_1 \mapsto u_1\,,\ \mathsf{Inr}\, x_2 \mapsto u_2\}) : (T_1 \oplus T_2) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-PatP-Foc}\ \frac{m\cdot\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x_1 :_m T_1,\ x_2 :_m T_2 \vdash u : U}{\Delta_1 \dashv C \circ (\square \triangleright \mathbf{case}_m\ (x_1\,, x_2) \mapsto u) : (T_1 \otimes T_2) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-PatE-Foc}\ \frac{m\cdot\Delta_1,\ \Delta_2 \dashv C : U \rightarrowtail U_0 \qquad \Delta_2,\ x :_{m\cdot m'} T \vdash u : U}{\Delta_1 \dashv C \circ (\square \triangleright \mathbf{case}_m\ \mathsf{E}_{m'}\ x \mapsto u) : !_{m'} T \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-Map-Foc}\ \frac{\Delta_1,\ \Delta_2 \dashv C : U \ltimes T' \rightarrowtail U_0 \qquad 1^{\uparrow}\cdot\Delta_2,\ x :_{1^{\nu}} T \vdash t' : T'}{\Delta_1 \dashv C \circ (\square \triangleright \mathbf{map}\ x \mapsto t') : (U \ltimes T) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-ToA-Foc}\ \frac{\Delta \dashv C : (U \ltimes 1) \rightarrowtail U_0}{\Delta \dashv C \circ (\mathbf{to}_{\ltimes}\ \square) : U \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FromA-Foc}\ \frac{\Delta \dashv C : (U \otimes (!_{1^{\infty}} T)) \rightarrowtail U_0}{\Delta \dashv C \circ (\mathbf{from}_{\ltimes}\ \square) : (U \ltimes (!_{1^{\infty}} T)) \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillU-Foc}\ \frac{\Delta \dashv C : 1 \rightarrowtail U_0}{\Delta \dashv C \circ (\square \triangleleft ()) : \lfloor_n 1 \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillL-Foc}\ \frac{\Delta \dashv C : \lfloor_n T_1 \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ (\square \triangleleft \mathsf{Inl}) : \lfloor_n T_1 \oplus T_2 \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillR-Foc}\ \frac{\Delta \dashv C : \lfloor_n T_2 \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ (\square \triangleleft \mathsf{Inr}) : \lfloor_n T_1 \oplus T_2 \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillP-Foc}\ \frac{\Delta \dashv C : (\lfloor_n T_1 \rfloor \otimes \lfloor_n T_2 \rfloor) \rightarrowtail U_0}{\Delta \dashv C \circ (\square \triangleleft (,)) : \lfloor_n T_1 \otimes T_2 \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillE-Foc}\ \frac{\Delta \dashv C : \lfloor_{m\cdot n} T \rfloor \rightarrowtail U_0}{\Delta \dashv C \circ (\square \triangleleft \mathsf{E}_m) : \lfloor_n !_m T \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillF-Foc}\ \frac{\Delta_1,\ (1^{\uparrow}\cdot n)\cdot\Delta_2 \dashv C : 1 \rightarrowtail U_0 \qquad \Delta_2,\ x :_m T \vdash u : U}{\Delta_1 \dashv C \circ (\square \triangleleft (\lambda x_{\,m}\!\mapsto u)) : \lfloor_n T \multimap_m U \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillComp-Foc1}\ \frac{\Delta_1,\ (1^{\uparrow}\cdot n)\cdot\Delta_2 \dashv C : T \rightarrowtail U_0 \qquad \Delta_2 \vdash t' : U \ltimes T}{\Delta_1 \dashv C \circ (\square \triangleleft\!\bullet t') : \lfloor_n U \rfloor \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-FillComp-Foc2}\ \frac{\Delta_1,\ (1^{\uparrow}\cdot n)\cdot\Delta_2 \dashv C : T \rightarrowtail U_0 \qquad \Delta_1 \vdash v : \lfloor_n U \rfloor}{\Delta_2 \dashv C \circ (v \triangleleft\!\bullet \square) : U \ltimes T \rightarrowtail U_0}$$

$$\textsc{Ty-ectxs-OpenAmpar-Foc}\ \frac{\begin{array}{c}\mathtt{LinOnly}\ \Delta_3 \quad \mathtt{FinAgeOnly}\ \Delta_3 \\ \mathbf{hnames}(C)\ \#\#\ \mathbf{hnames}(\Delta_3) \\ \Delta_1,\ \Delta_2 \dashv C : (U \ltimes T') \rightarrowtail U_0 \\ \Delta_2,\ \rightarrow^{-1}\Delta_3 \overset{\vee}{\vdash} v_2 : U\end{array}}{1^{\uparrow}\cdot\Delta_1,\ \Delta_3 \dashv C \circ ({}^{\mathrm{op}}_{\mathbf{hnames}(\Delta_3)}\langle v_2 \wedge \square\rangle) : T' \rightarrowtail U_0}$$

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

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

Fig. 17. Typing rules for evaluation contexts and commands

- reduction rules, where the actual computation logic takes place.

Here is the whole set of rules for PatP:

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

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

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

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 18.

The 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 $\mathrm{C}[h :=_H \mathrm{v}]$ 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 $h' = \mathbf{max}(\mathbf{hnames}(\mathrm{C}) \cup \{h\}) + 1$ in the premises of these rules. The implicit invariant of the system is that we always rename hole names inside an ampar to fresh names when that ampar is **map**ped on. Names bound by a closed ampar are only local, and thus can shadow already existing names (that's why **alloc** is allowed to reuse the same hole name 1 at each invocation: it doesn't matter as it desugars to a closed ampar). Thus we only need to take into account the names of already opened ampars — all part of the evaluation context — when we want fresh names (either for new holes, or for ampar renaming).

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 $[H \doteq h']$ on v_2 and v_1 (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 ${}^{\mathrm{op}}_{H \doteq h'}\langle \mathrm{v}_2 [H \doteq h'] \wedge \square \rangle$ 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.

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

Sem-App-Red

$$C[(\lambda^{\gamma} x_{\,m} \mapsto u)\ v] \longrightarrow C[u[x := v]]$$

Sem-PatU-Red

$$C[() \fatsemi u] \longrightarrow C[u]$$

Sem-PatL-Red

$$C[(\mathsf{Inl}\, v_1) \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\, x_1 \mapsto u_1,\ \mathsf{Inr}\, x_2 \mapsto u_2\}] \longrightarrow C[u_1[x_1 := v_1]]$$

Sem-PatR-Red

$$C[(\mathsf{Inr}\, v_2) \triangleright \mathbf{case}_m\ \{\mathsf{Inl}\, x_1 \mapsto u_1,\ \mathsf{Inr}\, x_2 \mapsto u_2\}] \longrightarrow C[u_2[x_2 := v_2]]$$

Sem-PatP-Red

$$C[(v_1, v_2) \triangleright \mathbf{case}_m\ (x_1, x_2) \mapsto u] \longrightarrow C[u[x_1 := v_1][x_2 := v_2]]$$

Sem-PatE-Red

$$C[\mathsf{E}_n\, v' \triangleright \mathbf{case}_m\ \mathsf{E}_n\, x \mapsto u] \longrightarrow C[u[x := v']]$$

Sem-ToA-Red

$$C[\mathbf{to}_{\ltimes}\, v_2] \longrightarrow C[\{\}\langle v_2 \wedge ()\rangle]$$

Sem-FromA-Red

$$C[\mathbf{from}_{\ltimes}\ \{\}\langle v_2 \wedge \mathsf{E}_{1\infty}\, v_1\rangle] \longrightarrow C[(v_2, \mathsf{E}_{1\infty}\, v_1)]$$

Sem-FillU-Red

$$C[\to h \triangleleft ()] \longrightarrow C[h :=_{\{\}}\ ()][()]$$

Sem-FillF-Red

$$C[\to h \triangleleft (\lambda x_{\,m} \mapsto u)] \longrightarrow C[h :=_{\{\}}\ \lambda^{\gamma} x_{\,m} \mapsto u][()]$$

Sem-FillL-Red

$$\frac{h' = \mathbf{max}(\mathbf{hnames}(C) \cup \{h\}) + 1}{C[\to h \triangleleft \mathsf{Inl}] \longrightarrow C[h :=_{\{h'+1\}}\ \mathsf{Inl}\, \boxed{h'+1}][\to h' + 1]}$$

Sem-FillR-Red

$$\frac{h' = \mathbf{max}(\mathbf{hnames}(C) \cup \{h\}) + 1}{C[\to h \triangleleft \mathsf{Inr}] \longrightarrow C[h :=_{\{h'+1\}}\ \mathsf{Inr}\, \boxed{h'+1}][\to h' + 1]}$$

Sem-FillE-Red

$$\frac{h' = \mathbf{max}(\mathbf{hnames}(C) \cup \{h\}) + 1}{C[\to h \triangleleft \mathsf{E}_m] \longrightarrow C[h :=_{\{h'+1\}}\ \mathsf{E}_m\, \boxed{h'+1}][\to h' + 1]}$$

Sem-FillP-Red

$$\frac{h' = \mathbf{max}(\mathbf{hnames}(C) \cup \{h\}) + 1}{C[\to h \triangleleft (,)] \longrightarrow C[h :=_{\{h'+1, h'+2\}}\ (\boxed{h'+1}, \boxed{h'+2})][(\to h' + 1, \to h' + 2)]}$$

Sem-FillComp-Red

$$\frac{h' = \mathbf{max}(\mathbf{hnames}(C) \cup \{h\}) + 1}{C[\to h \triangleleft\!\bullet_H \langle v_2 \wedge v_1\rangle] \longrightarrow C[h :=_{(H \pm h')}\ v_2[H \pm h']][v_1[H \pm h']]}$$

Sem-Map-Red-OpenAmpar-Foc

$$\frac{h' = \mathbf{max}(\mathbf{hnames}(C)) + 1}{C[{}_H\langle v_2 \wedge v_1\rangle \triangleright \mathbf{map}\ x \mapsto t'] \longrightarrow (C \circ ({}^{\mathrm{op}}_{H \pm h'}\langle v_2[H \pm h'] \wedge \Box\rangle))[t'[x := v_1[H \pm h']]]}$$

Sem-OpenAmpar-Unfoc

$$(C \circ^{\mathrm{op}}_H \langle v_2 \wedge \Box\rangle)[v_1] \longrightarrow C[{}_H\langle v_2 \wedge v_1\rangle]$$

Fig. 18. Small-step semantics

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_1 to reform a closed ampar.

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

Theorem 7.1 (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 7.2 (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.

8 REMARKS ON THE FORMAL PROOFS

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

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.

a citation maybe

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 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.

citation

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.

9 IMPLEMENTATION OF DESTINATION CALCULUS USING IN-PLACE MEMORY MUTATIONS

The formal language presented in Sections 5 to 7 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. $\mathsf{E}_{\omega\nu}\,\{h\}\langle \mathsf{Inr}\,(\mathsf{Inl}\,()\,,\,\boxed{h})\,,_{\wedge} \to h\rangle$ (representing a non-linear difference list $\mathsf{E}_{\omega\nu}\,(0 :: \square)$), and then used twice, each time in a different way:

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

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 7.3, 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):

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

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 $\mathbf{alloc_{ip}}$. 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 $\mathbf{alloc_{ip}}$, that we call $\lambda_{d\,\mathbf{ip}}$, "closed" programs now typecheck in the non-empty context $\{tok_0 :_{1\omega} \mathsf{Token}\}$ 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 $\{tok_0 :_{1\omega} \mathsf{Token}\}$ is very similar to having a primitive function **withToken** : $(\mathsf{Token} \to_{1\omega} !_{\omega\omega}\mathsf{T}) \to !_{\omega\omega}\mathsf{T}$ as it is done in [Bagrel 2024].

In $\lambda_{d\,\mathbf{ip}}$, 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 fresh freshness/renaming 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" ($\blacktriangleleft$) 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.

10 RELATED WORK

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

10.1 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.

10.2 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.

10.3 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 $S \ltimes T$, only the right hand part.

In [Minamide 1998], it's only ever possible to represent structures with a single hole. But this is a rather superficial restriction. The author doesn't comment on this, but we believe that this restriction only exists for convenience of the exposition: the language is lowered to a language without function abstraction and where composition is performed by combinators. While it's easy to write a combinator for single-argument-function composition, it's cumbersome to write combinators for functions with multiple arguments. But having multiple-hole data structures wouldn't have changed their system in any profound way.

The more important difference is that while their system is based on a type of linear functions, our is based on the linear logic's par combinator. This, in turns, lets us define a type of destinations which are representations of holes in values, which [Minamide 1998] doesn't have. This means that [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.

10.4 Tail modulo context

You'll [Thomas] have to do this section, because I'm not familiar enough with that work

10.5 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.

10.6 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 $\lfloor \mathsf{T} \rfloor$ 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.

11 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 an 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